POLITECNICO DI TORINO

Master's Degree in Ingegneria Matematica

Master's Degree Thesis

Simulation-Based Optimization of On-Orbit Refueling Mission Scheduling for Satellites.

Supervisors

Prof. Edoardo FADDA

Candidate

Andrea BACCOLO

Academic Year 2024-2025

A Claudia, Roberto e Silvana

Abstract

The increasing demand for satellite servicing and the high cost involved in keeping them operative has led on-orbit refueling to be considered one of the most effective ways to extend satellite lifespan and mission flexibility. This thesis presents a simulation-based optimization framework for scheduling refueling missions for satellites in geosynchronous orbits. The optimization problem addresses the travel mission of multiple Service Spacecraft from an on-orbit station to multiple target satellites that need refueling. The mission objective is to find the optimal sequence of targets to refuel for every Service Spacecraft, minimizing the total fuel consumption during all rendezvous. Firstly, a simulator that computes and executes the maneuvers is proposed. The optimization method used is the Adaptive Large Neighborhood Search Heuristic, this method uses some proposed "destroy" and "repair" operators to generate new solutions. Finally, some experiments using a fixed scenario are performed to explore various strategies and evaluate different operators.

Table of Contents

List of Tables

List of Figures

Acronyms

ACO Ant Colony Optimization

AGA Adaptive Genetic Algorithm

ALNS Adaptive Large Neighborhood Search

EA Evolutionary Algorithms

FS Fuel Station

GA Genetic Algorithm

LNS Large Neighborhood Search

OOS On-Orbit Service

PSO Particle Swarm Optimization

RAAN Right Ascension of the Ascending Node

SA Simulated Annealing

SSc Service Spacecraft

Chapter 1

Introduction

A relevant characteristic of satellites is self-sufficiency and robustness, however, their limited lifespan have usually been shortened by factors like fuel consumption, component failure, or technical obsolescence and, as a result, this often led to deactivation and expanding the space debris issue. In this inefficient and economically unsustainable setting, On-Orbit Service (OOS) has become a fundamental tool to mitigate some of those problems. Generally, OOS exploits some smaller satellites, called Service Spacecraft (SSc), to offers one or more services to some satellites usually called targets. Those kind of missions might also avoid replacing the target, decreasing the probability of the mission's failure and saving money on lower launch costs [1]. In recent years, OOS missions have developed drastically [2] and a large number of researchers have investigated the OOS mission planning problem for multiple satellites. Those missions can be categorized into four main groups according to the kind of mission they address: On-orbit Repairing, Debris removal missions, On-orbit Refueling and Mixed missions. *Repairing missions* are crucial to extend the lifespan of satellites. [3] proposed a many to many mission planning considering time and velocity constraints. *Debris removal missions* try to decrease the likelihood of collisions by reducing space debris. The principal difference between debris removal and other OOS missions is that, upon a rendezvous with the target, the SSc must transport the target to graveyard orbit and release it. An example of Debris removal mission is shown in fig. 1.1. [4] addressed this problem using a single SSc and discovering some heuristic laws. [5] suggested two types of debris mission: multi and single SSc, while also proposing two transfer models: a static and a dynamic one. [6] considers both chemical and electric propulsion in task allocation , while [7] provided a multi objective optimization, trying to minimize the total velocity used in rendezvous while choosing debris to remove that maximize a measure of danger. *On-orbit refueling missions* consider a set of SScs that need to refuel a set of targets. Usually, one or more Fuel Station (FS) are employed so that the SScs can take some fuel and bring it to the targets.

Figure 1.1: An example of a multiple GEO space debris removal process [6]
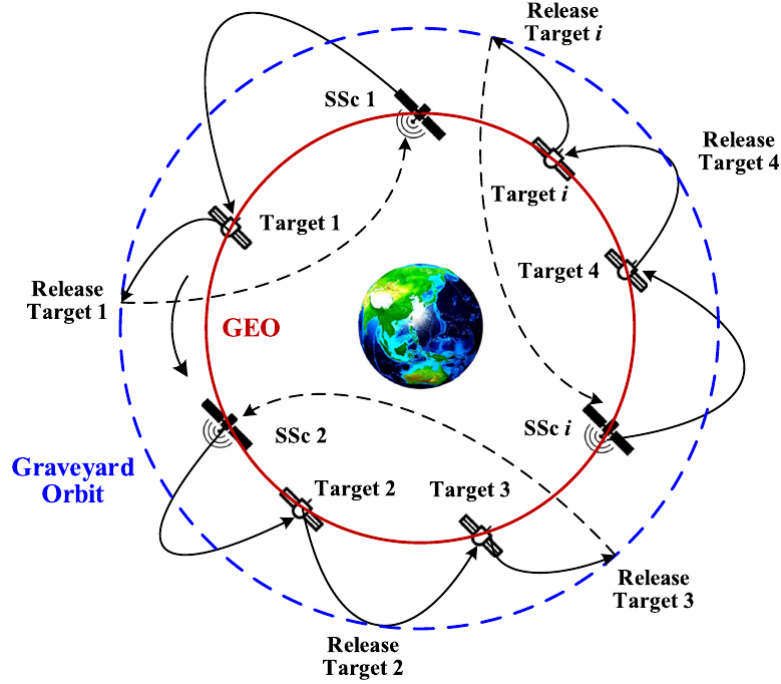
That allows the SSc to be lighter and use less fuel. When an SSc reaches the target, it refills it and then rendezvous with another target or return to some FS. An SSc will complete a mission (in this work, also called tour) when it leaves the FS, refills all the assigned targets and returns to the FS. A refueling mission scenario is presented in fig. 1.2. One may argues that OOS can't be useful for satellites with electric propulsion, however, as stated in [8], electrical propulsion is an important development mode, but cannot completely replace chemical propulsion. [8] proposed the "$1 + N$" architecture adopted in this work. This architecture is formed by N SScs and a single FS that carries a sufficient quantity of fuel for all the missions. The first kind of those tanks has been launched in 2021 [9]. [10] adopted a one to many framework with one fuel station, while [11] used a many to many setting with different FSs. [12] considered the option of a closed or open FS. [13] suggested the option SScs of taking more than one tour and take into account uncertainty in refueling time. On the other hand, [14] worked on a totally different problem. In a one to many setting, all targets need to choose a single service position from a set of service positions. The SSc needs to choose the order of servicing position to reach, then return to it's original position. *Mixed missions* attempt to do several tasks on a single schedule. [15] addressed a multi objective optimization, minimizing both fuel cost and time, considering a general service performed to the target.

Figure 1.2: A visualization of a single refueling mission scenario [13]
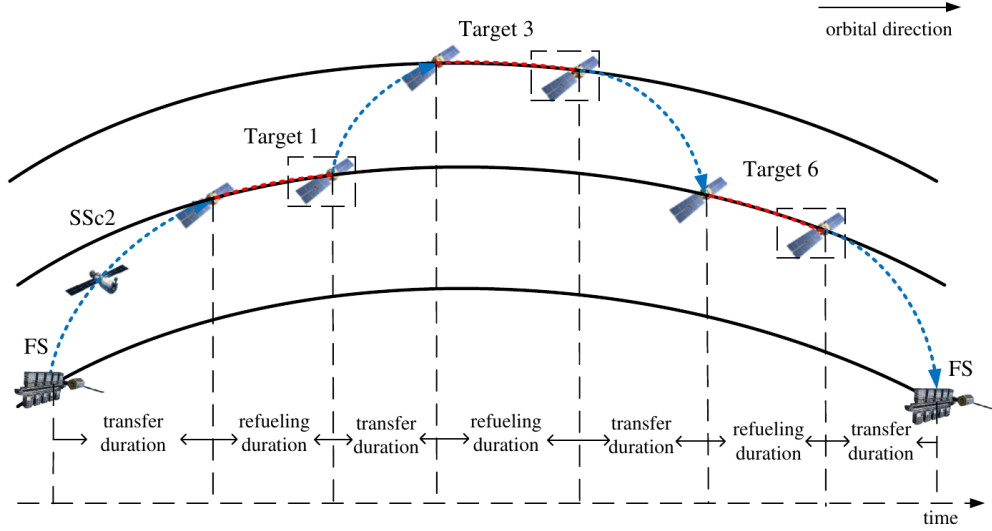
[16] considered the possibility of a target to become a "Pseudo SSc", which will be used to service other targets. Moreover, the dry mass of targets became a design variable, to better decrease the overall cost of the mission. A noticeable disadvantage of the mixed OOS, as stated in [3], is the fact that each SSc must carry a variety of object to meet different OOS services, this causes a high dry weight and consequently higher fuel consumption during orbital maneuvers. Moreover, [8] compared three orbit deployments based on response time and economic benefit, proposing the "*1+N*" architecture explained before. Another interesting application is in *Sun-synchronous orbits*, studied in [17]. Regarding optimization, basically all selected literature uses meta-heuristic. The only two exceptions are [18], which used branch and bound, exploiting integer linear programming and mixed integer non linear programming formulations to manage feasibility and tours; and [4], which developed a "rapid method" based on the heuristic laws that they discovered. Among the meta-heuristics, two important Swarm-Based meta-heuristics are used: Ant Colony Optimization (ACO) in [12], and Particle Swarm Optimization (PSO) in [16]. In addiction, [11] used a hybrid version of PSO combined with exhaustive search, while [15] employed a multi-Objective version of PSO. [3] adopted a Large Neighborhood Search (LNS) framework combined with Adaptive Genetic Algorithm (AGA). All other meta-heuristiucs used are Evolutionary Algorithms (EA), which are inspired by evolutionary principles such as re-combinations, mutation and selections. [6] proposed a clustering based adaptive differential evolution algorithm with potential individual reservation, [2] proposed an intelligent global optimization algorithm called evolving elitist club algorithm, that has been compared to

ACO. [7] used four multi-objective EA: Multi-Objective EA based on Decomposition, Strength Pareto EA 2, Non-dominated sorting Genetic Algorithm (GA) II, Multi-Objective PSO. [13] proposed a hybrid Harris Hawks Algorithm combined with a crossover strategy with GA. Lastly, some GA were also used: [14] utilized hybrid encoding GA, [10] adopted GA combined with random search and [17] used genetic quantum algorithms. This work addressed the problem of scheduling a refueling mission with a Adaptive Large Neighborhood Search (ALNS) framework, and it is organized as followed: Chapter 2 briefly explains some prior knowledge on the astronomical setting, describing orbital elements and maneuvers, to give a proper background and context. Chapter 3 defines the refueling process and constructs the optimization model. Chapter 4 shows an ALNS framework, while Chapter 5 explains some implemented destroy and repair operators. Chapter 6 gives and evaluates a simulation examples, comparing operators and highlighting some interesting discovery. Finally, Chapter 7 summarizes conclusions and some possible future research directions.

Chapter 2

Astrodynamics background

2.1 Kepler's laws

Approximately 400 years ago, Kepler used Tycho Brahe's documented observations to empirically check the laws of planetary motion. Those laws are still widely use for modeling and calculating important quantities such as the period of an orbit or the velocity of a body on that orbit. Kepler's first law states that each planet follows an elliptical orbit around the Sun. The Sun's position is located in one of the foci of the ellipse. Kepler was able to show that with the Sun at one of the foci, every planet traces an elliptic path around the center of the ellipse ([19] chapter 1.2). Figure 2.1 illustrates the concept of Kepler's first laws, the orbit has the following equation:

$$r(\nu) = \frac{a(1 - e^2)}{1 + e \ cos(\nu)}.$$

(2.1)

 The quantity a (green in fig. 2.1) and e are the *semi-major axis* and the *eccentricity* respectively. e measures how far an orbit deviates from a circle: a perfect circle is obtained with $e = 0$, while for $e < 1$ correspond to an elliptic orbit ([20], chapter 2.1.2)). The closest point to a specific focus is called peri-apsis, on the other hand the furthest point is called apo-apsis. Using a polar coordinate system, as shown in fig. 2.1, the peri-apsis is collocated at $(r = a \ (1 - e), \nu = 0)$ while the apo-apsis is at $(r = a \ (1 + e), \nu = \pi)$ (orange in fig. 2.1). According to Kepler's second law, the straight line connecting the Sun and a planet sweeps out equal areas in equal amounts of time. This implies that on an elliptic orbit, the velocity is not constant, thus is faster on the peri-apsis and slower on the apo-apsis. Finally, Kepler's remarkable third law connects the shape of an orbit with it's orbital period: The cubes of the semi-major axes of the planet's orbits are proportional to the squares

Figure 2.1: Kepler's first law
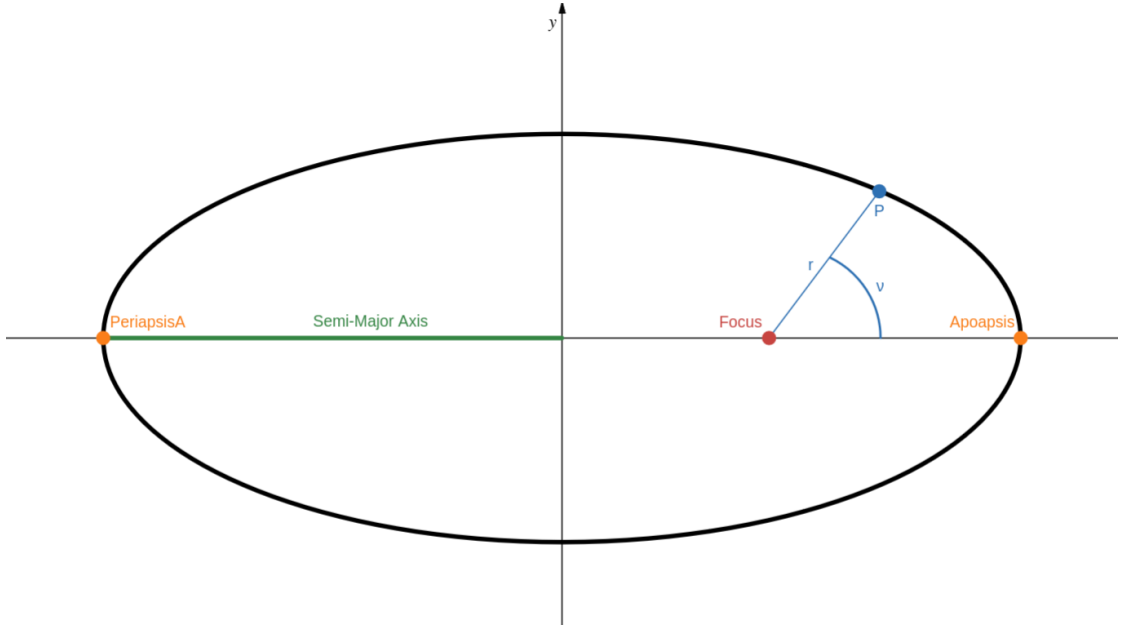
of their orbital periods. In fact:

$$a^3 = T^2 \, \frac{G \cdot (M + m)}{4 \, \pi^2}. \tag{2.2}$$

T is the orbital period of the satellite, $G = 6.6725910^{-11} \frac{m^3}{Kg \; s^2}$ is the gravitational constant, M is the mass of the body on the focus while m is the mass of the orbiting body. In this work, the body in the focus will always be the Earth and the orbiting body will be a satellite. Since the mass of a satellite is negligible if compared to the mass of the Earth, the following approximation will be used:

$$G \cdot (M + m) \approx GM = 3.986004418 \cdot 10^5 \, \frac{km^3}{s^2} := \mu. \tag{2.3}$$

This is known as the *Earth's gravitational constant*.

2.2 Classical orbital elements

Before explaining what orbital elements are, a suitable reference system is described [20], Section 2.2.3 and [19] Chapter 3.3. The coordinate system used is based on the Earth's orbit around the Sun. The plane of the Earth's mean orbit around the sun is called *ecliptic*. When an orbit creates an intersection with a plane, the resulting

points are known as *nodes*. Those nodes are called *ascending node*, when the object on the orbits travels that specific node from south to north, and *descending node* when it does from north to south. The intersections between the ecliptic and the equatorial plane are specifically called *equinoxes*: the ascending node corresponds to the spring equinox and it is called *vernal equinox*, the descending node corresponds to the fall equinox and it is called *autumnal equinox* (the seasons are for the Northern Hemisphere). Consider a coordinate system in which the x-y plane is formed by the equatorial plane and the z-axis points to the north pole. The x-axis passes through the vernal equinox, and the y-axis is obtained choosing the only possible direction perpendicular to the previous axes. It is important to remark that in this specific coordinate system, going from west to east is equivalent to a rotation on the z-axis from the x-axis to the y-axis, which follows the Earth rotation indicated by the z-axis using the right hand rule. As explained in [19], section 1.2.3, a general body on a specific orbit can be characterized by six parameters:

- a: *Semi-major axis*,

- e: *Eccentricity*,

- i: *Inclination*,

- ω: *Argument of perigee*,

- Ω: *Right Ascension of the Ascending Node (RAAN)*,

- ν: *True anomaly*,

The first two describes the form of the object's orbit, as already stated before. The *Inclination* is the angle from the z-axis to the angular momentum that describe the rotation on the orbit. It determines how much the orbit is inclined with respect the x-y plane. It ranges from 0 to 180 degrees. If an orbit has an inclination between 0 and 90 degrees, the body on that orbit will travel in the same direction of rotation as the Earth and the orbit is called *direct* or *pro-grade*. This means that in an orbit with an inclination inferior to 90 degrees a body moves counter clock wise from above. On the other hand, an inclination between 90 and 180 degrees means that the body on that orbit will travel in the opposite direction of Earth's rotation, the orbit is called *retrograde* orbit. If an orbit is inclined, it has a *line of nodes*, which is the intersection of the equatorial plane with the inclined plane in which the orbit resides. In other words, the line of nodes is the line connecting the two nodes obtained through the intersection of the orbit and the equatorial plane. The *RAAN* is the angle in the equatorial plane measured positive eastward from the x-axis to the location of the ascending node. The *Argument of perigee* is the angle between the direction of the ascending node and the perigee and it varies from 0 to 360 degrees. The *True anomaly* is an angle that describes the position of the body

on the orbit. It goes from 0 to 360 degrees and it is measured from the peri-apsis to the satellite. A better visualization is proposed in fig. 2.2 on a circular orbit case.

2.3 Circular orbits

In the considered model the orbits are geosynchronous and circular (assumption (A1)). This approximation helps to simply the model, one of the advantages is considering three orbital elements instead of six to characterize a point on a specific orbit: the true anomaly, inclination and RAAN. However, the true anomaly must be redefined since on an circular orbit, the peri-apsis does not exist. For an equatorial orbit, the true anomaly has been defined as the angle measured from the x-axis, positive in the direction of rotation of the orbit. For an inclined orbit, the starting point has been chosen by simply applying the rotations of the orbit to the x-axis vector. The two rotation applied are a rotation on the x-axis for inclination and z-axis for the RAAN. This will be important when calculating the true anomaly of a specific tridimensional point in Section 2.4 for Subsection 2.5.1. In the considered model, it can happens that different circular orbital element may

Figure 2.2: A graphic visualization of RAAN Ω (blue), inclination i (purple), true anomaly (red) obtained with Desmos.
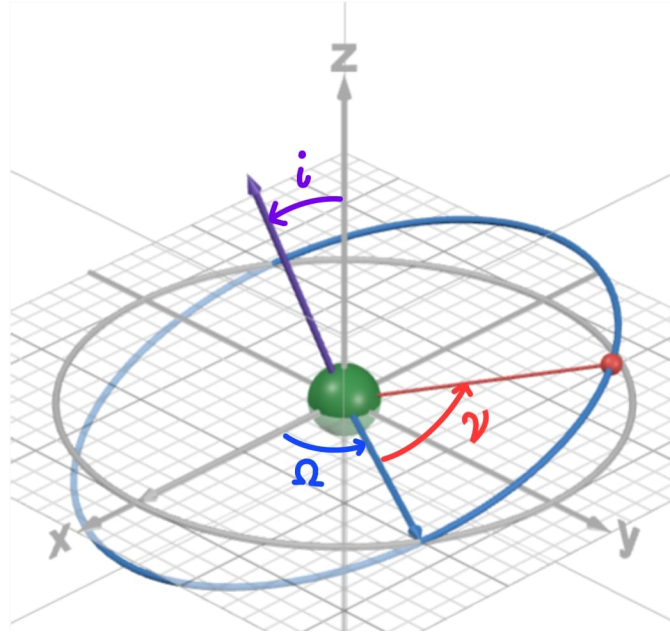
describe the same path. Figure 2.3 exposes this scenario: the orange orbit has $(i = 130, \Omega = 60)$, while the blue orbit has $(i = 50, \Omega = 240)$. The orange and the

Figure 2.3: An example of orbits with different circular orbital element but same trajectory.
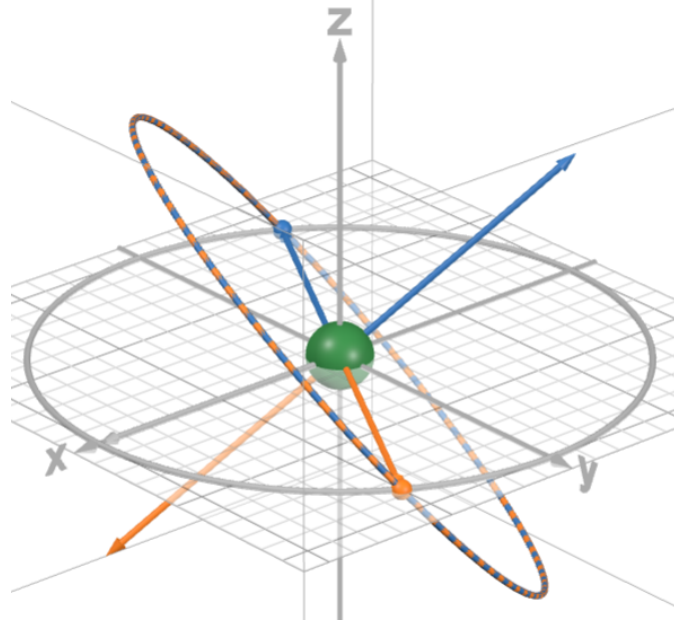
blue vector perpendicular to the orbital plane are the angular momentum of the orbit of the respective color. This occurs when the following criteria are met:

$$\begin{cases} i_1 + i_2 = 180, & \text{(2.4a)} \\ |\Omega_1 - \Omega_2| = 180. & \text{(2.4b)} \end{cases}$$

When the condition of eq. (2.4a) is satisfied and $\Omega_1 = \Omega_2$, the two circles are one the reflection of the other with respect the z-x plane rotated by Ω_1 over the z-axis. Moreover, the two angular momentum are one the reflection of the other with respect the x-y plane. As a result of adding the condition of eq. (2.4b), the two angular momentum become one the point reflection of the other with respect the origin, therefore the two vector will be parallel. The two couple of orbital element describe the same orbit, but the directions of rotation are opposite. In this setting, the planar change is harder, since in a real-world scenario proximity maneuvers require the same direction of rotation between the reacher and the target. Those cases are not considered in the model (assumption (A6)). Every instances are checked before created using an implemented function called *checkInstance*, more detail on the Test folder of the project (appendix B).

2.4 Rotation Matrices

During the simulation it will be important to obtain the true anomaly from a specific tridimensional point on the orbit with respect the coordinate system explained before. This has been accomplished using some rotation matrices. In fact, this task has been divided in two different steps: rotating the point from an inclined orbit to an equatorial one and getting the angle. Starting from [20] section 2.2.3, using some algebraic multiplication, it can be shown the following:

$$\mathbf{P}_{eq} = \boldsymbol{R}_y(\omega)\, \boldsymbol{R}_x(i)\, \boldsymbol{R}_z(\Omega)\, \mathbf{P}_{in}, \tag{2.5}$$

where $\mathbf{P}_{in}$ is the input tridimensional point, $\mathbf{P}_{eq}$ is the equatorial point and

$$\boldsymbol{R}_x(\phi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & +\cos\phi & +\sin\phi \\ 0 & -\sin\phi & +\cos\phi \end{pmatrix} \quad \boldsymbol{R}_y(\phi) = \begin{pmatrix} +\cos\phi & 0 & +\sin\phi \\ 0 & 1 & 0 \\ -\sin\phi & 0 & +\cos\phi \end{pmatrix} \tag{2.6}$$

$$\boldsymbol{R}_z(\phi) = \begin{pmatrix} +\cos\phi & +\sin\phi & 0 \\ -\sin\phi & +\cos\phi & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{2.7}$$

are rotation matrices with respect its specific axis. It is worth noticing that in the circular case, the rotation on the y-axis is not used. Since the argument of perigee is not define, it can be permanently set to zero, obtaining the identity matrix from $\boldsymbol{R}_y(\phi)$. The second step starts by noticing that a generic point on a circle can be written in the following form:

$$\mathbf{P} = r \begin{pmatrix} \cos v \\ \sin v \\ 0 \end{pmatrix}, \tag{2.8}$$

where r is the radius of the considered circle. Thus the true anomaly can be found by solving the following system:

$$\begin{cases} \nu = \cos^{-1}\left(\dfrac{P_{eq,x}}{r}\right) \\ \nu = \sin^{-1}\left(\dfrac{P_{eq,y}}{r}\right). \end{cases} \tag{2.9}$$

2.5 Maneuvers

An SSc needs to perform some maneuvers in order to reach a target. Theoretically, due to Earth's gravitational force, the satellite does not need to use fuel to stay

on one orbit, however the Earth is not a perfect sphere and, since it's surface is not at a fixed altitude, the gravitational potential changes slightly creating some perturbations. To maintain the orbit, some station-keeping maneuvers are required in a real setting. Those variations are not considered in the model (assumption (A5)), still some impulse is requested for change one or more orbital element. A variation of velocity has to be applied, from this variation (called from now on Δv) depends the amount of fuel used during the maneuver. Fuel consumption is modeled using *Tsiolkovsky rocket equation*, which has the following form:

$$m_f = m_i \, e^{-\frac{\Delta v}{g \, I_{sp}}}, \tag{2.10}$$

where m_f and m_i are the final and initial mass respectively, g is the gravitational constant. I_{sp} is the Specific Impulse, an information quantity about the efficiency of the rocket. Equation (2.10) shows that the mass after the maneuver is the initial mass "discounted" by a factor that depends of the variation of velocity employed and the efficiency of the rocket. After calculating the theoretical mass obtained after the maneuver, the quantity of fuel used in the maneuver can be computed by the following difference: $f = m_i - m_f$. If the fuel quantity computed this way is inferior to the fuel mass of the SSc, then the maneuver can be performed. On the proposed simplified model, two types of maneuvers has been implemented: the planar change and the phasing. Typically some proximity maneuvers are required after the phasing to get closer to the target with safety. The model does not take them into account (assumption (A5)).

2.5.1 Planar Change

In this subsection the planar change maneuver from [3] will be explained. The goal of this maneuver is to change the SSc's orbit into the target one. In other words, the maneuver change the inclination and the RAAN to get in the same target's orbit. This particular maneuver can only be performed by applying a single impulse at one of the intersection of the two orbits (that corresponds to nodes only if one of the two orbits is equatorial). A visualization is proposed in fig. 2.4. Let ϑ_{pc} be the *dihedral angle* between two orbital plane, an important angle that will be used to calculate the variation of velocity needed. It holds the following equation:

$$\cos \vartheta_{pc} = \sin I_{\mathrm{s}} \sin I_{\mathrm{t}} \cos \left(\Omega_{\mathrm{s}} - \Omega_{\mathrm{t}} \right) + \cos I_{\mathrm{s}} \cos I_{\mathrm{t}}, \tag{2.11}$$

where $I_{\mathrm{s}}, I_{\mathrm{t}}, \Omega_{\mathrm{s}}, \Omega_{\mathrm{t}}$ are the inclination and RAAN of the SSc and target, respectively. The two intersection point are computed in the following way:

$$\begin{cases} \boldsymbol{r}_{\mathrm{m1}} = a \frac{\mathbf{h_s} \times \mathbf{h_t}}{\|\mathbf{h_s} \times \mathbf{h_t}\|} \\ \boldsymbol{r}_{\mathrm{m2}} = -\boldsymbol{r}_{\mathrm{m1}} \end{cases} \tag{2.12}$$

11

Figure 2.4: A Planar change Maneuver [13]
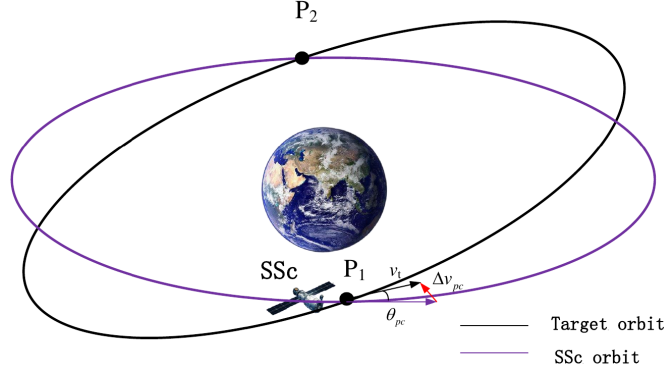
where a is the radius of the GEO orbit. $\mathbf{h_t}$ and $\mathbf{h_s}$ are the angular momentum of the target and SSc respectively. The general angular momentum of an orbit can be calculated using the following formula:

$$\boldsymbol{h} = \sqrt{\mu a} \begin{bmatrix} \cos\Omega & -\sin\Omega & 0 \\ \sin\Omega & \cos\Omega & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos i & -\sin i \\ 0 & \sin i & \cos i \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \qquad (2.13)$$

where μ is the Earth's gravitational constant mentioned in Section 2.1, Ω is the RAAN and i is the inclination. It is worth noticing that $\boldsymbol{h}$ is constant, because the satellite cannot leave the orbital plane since the gravitational force is always anti-parallel to the position vector. Due to this fact it does not give any acceleration perpendicular to the plane ([20], section 2.1.1). In the case of planar change, since we are dealing with GEO orbits (assumption (A1)), the variation of velocity is to be intended just as a variation of directions. Let $\boldsymbol{v}_\text{t}$ and $\boldsymbol{v}_\text{s}$ be the velocity vectors centered in the chosen intersection of the target and SSc respectively. The variation and it's norm can be computed in the following:

$$\Delta\boldsymbol{v} = \boldsymbol{v}_\text{t} - \boldsymbol{v}_\text{s}, \qquad \|\Delta\boldsymbol{v}\| = 2\|\boldsymbol{v}_\text{t}\|\sin(\alpha/2). \qquad (2.14)$$

The norm will be used in eq. (2.10), moreover $\sin(\alpha/2)$ is computed from the $\cos\alpha$ using the bisection formula, considering only the positive sign since a norm is always positive. The decision of which nodes should be employed by the maneuver is made through a greedy policy: the selected node is the nearest to the SSc with respect to the rotation sense on the orbit.

2.5.2 Phasing

After the Planar change maneuver, the only SSc's orbital element that may differs from the target's one is the true anomaly. The phasing maneuver consists of

applying two impulses. The first impulse allows the SSc to change it's orbit so that it's velocity can increase or decrease. To end the maneuver, a second impulse makes the return of the SSc to the target's orbit possible. This section is taken from [3] and [19] Section 6.6.1. The most impacting quantity to this maneuver is the phase difference between the two satellites. Let θ be the *phasing angle* between the SSc and the target expressed in degrees defined in [3], it is computed in the following way:

$$\theta = \begin{cases} \rho, & |\rho| \leqslant 180 \\ -2\pi + |\rho|, & \rho > 180 \\ 2\pi - |\rho|, & \rho < -180 \end{cases} \tag{2.15}$$

where $\rho = (\Omega_s + \nu_s) - (\Omega_t + \nu_t)$, $\Omega_s, \Omega_t, \nu_s, \nu_t$ are the RAAN and true anomaly of the target and SSc respectively. One may ask why the RAAN appears in the formula since the initial orbit is the same for the SSc and the target. This definition will be later used in Section 5.1, and can also be applied in this setting since the two RAANs cancel out. As shown in fig. 2.5 and fig. 2.6, the phase angle define how the SSc reaches the target. If $\theta < 0$ (fig. 2.5), then the target is in front of the SSc with respect the direction of rotation on the orbit. To reach the target, the SSc will increase its angular velocity by switching into a smaller orbit. On the other hand, if $\theta > 0$ (fig. 2.6), then the target is behind the SSc with respect the direction of rotation on the orbit and the SSc will decrease its angular velocity by switching into a bigger orbit. The first quantity to compute is the time maneuver,

Figure 2.5: Phasing when $\theta < 0$ [19]   **Figure 2.6:** Phasing when $\theta > 0$ [19]
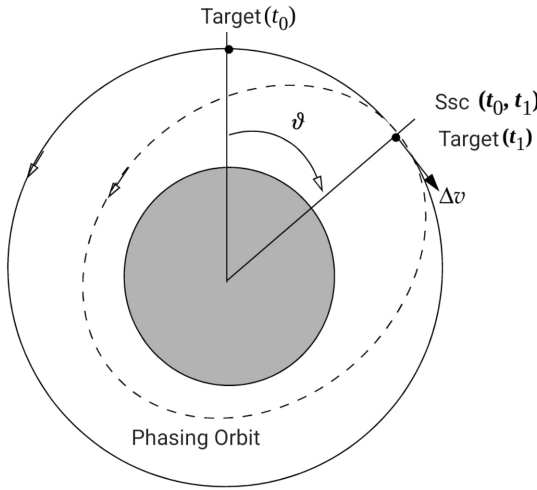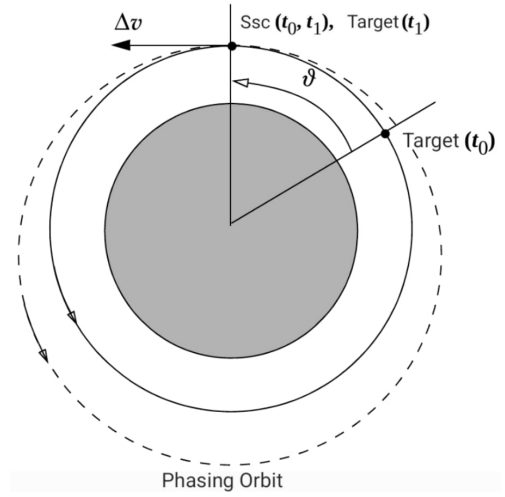
using the following equation:

$$\tau_{phase} = \frac{2\pi k_t + \theta \cdot \frac{\pi}{180}}{\omega}, \tag{2.16}$$

13

k_t is the number of revolutions that the target accomplished during the maneuver, ω is the angular velocity express in radians over seconds of the GEO orbit and the term $\frac{\pi}{180}$ is used to convert the degrees in radians. Moreover, the semi-major axis of the phasing orbit can be calculated as follows:

$$a_{phase} = r_{\text{GEO}} \left(\frac{2\pi k_{\text{t}} + \theta \cdot \frac{\pi}{180}}{2\pi k_i} \right)^{2/3} \tag{2.17}$$

k_i is the number of revolutions that the SSc accomplished during the maneuver. Since a_{phase} is computed after the time, it could happens that this quantity can be inferior to the radius of the Earth. A safe radius is implemented to check if the perigee is too close to the Earth. The total Δv is computed:

$$\Delta v = 2\sqrt{\mu} \left| \sqrt{\frac{2}{r_{\text{GEO}}} - \frac{1}{a_{\text{phase}}}} - \sqrt{\frac{1}{r_{\text{GEO}}}} \right| \tag{2.18}$$

As stated in [3] at the end of section 2.2.2, when k_t is chosen, it must hold $k_t = k_i$ in order to obtain the minimum Δv. In this work, k_t has been set to the smallest possible feasible value. During calculations, it is incremented only if the mission is infeasible due to negative times or semi-major axis being too small.

14

Chapter 3

Problem Formulation

3.1 Refueling process

In this chapter, the setting used for the refueling mission in [13] will be explained in more details, stating all the assumptions made. As previously explained in Chapter 1 and as fig. 3.1 shows, it is adopted a "$1 + N$" architecture [8]. A mission

Figure 3.1: An example of a refueling scenario [13]
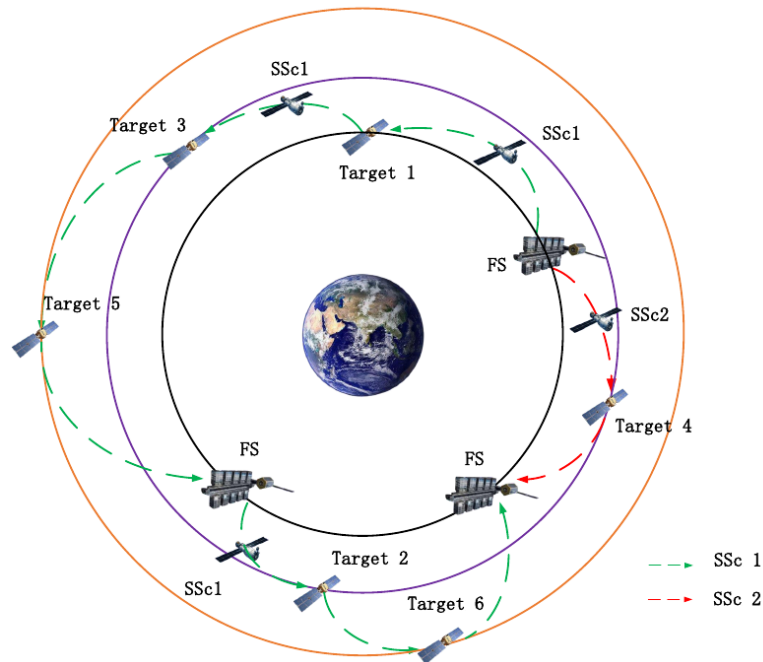
consists of a single SSc that leaves the station, refuels one or more targets of that

mission and returns to the station to be refueled and start the next mission. The operation is composed by two phases:

- *Planning*. Before leaving the FS, all targets are assigned to a single SSc in a specific order. Moreover, for every SSc the number of mission is chosen and the targets for every mission are determined.

- *Missions Execution*. Each SScs rendezvous with the target using maneuvers explained in Section 2.5 and give fuel to the target. If the SSc already visited all target's mission, it returns to the FS, otherwise it proceeds with the schedule.

Moreover, the refueling mission is studied under the following assumptions:

(A1) All orbits are circular and geosynchronous.

(A2) The FS possesses enough fuel for the entire operation.

(A3) The FS can refuel several SScs at the same time.

(A4) The simulation does not take into account the possibility of collisions between objects.

(A5) The simulation does not considers proximity maneuvers nor station-keeping maneuvers.

(A6) The case of same trajectory but different orbital elements discussed in Section 2.3 is not considered.

(A7) The increment of velocity is instantaneous.

(A8) The satellite's orientation is not take into account during maneuvers, only the Δv in modulus is considered.

(A9) The Planar change is istantaneus, as explained in Subsection 2.5.1.

(A10) The refueling process is deterministic, a speed of refueling is given for the FS and all SScs.

(A11) Every targets and SScs are refilled completely.

(A12) A target is refilled by only one SSc.

(A13) All targets possessed the same priority.

(A14) For every target, a single-target mission is always feasible, regardless of the initial position of the FS.

(A15) Removing one or more targets from a feasible mission would still produce a feasible mission.

Assumption (A14) is used to create a feasible initial solution in a fast way, by dividing the targets equally among the SScs and creating a mission for every one of them. Assumption (A15) is used during optimization, where feasibility is constantly check, for more detail, see Section 5.2.

3.2   Optimization Model

In this section, the optimization model will be formalized. Let $\mathcal{I} = \{1, \ldots, I\}$ be the set of SScs. Let $\mathcal{J} = \{1, \ldots, J\}$ be the set of Target and let $\mathcal{J}^* = \mathcal{J} \cup \{0\} = \{0, 1, \ldots, J\}$ be the set of the target with the FS, denoted by the zero. Every SScs will have μ_i missions denoted by the index $k \in \mathcal{K}_i = \{1, \ldots, \mu_i\}$. For every mission, a subset $\mathcal{J}_i^k \subseteq \mathcal{J}$ is chosen to denote which target need to be visit in mission k of the SSc i. Let $|\mathcal{J}_i^k| = n_{i,k}$ be the cardinality of that subset. The solution si typically provided by a matrix. The i-th row describes the path of SSc i. In table 3.1 an example of a possible solution with $J = 6$ and $I = 2$ is presented. Every cell of the sequence represents the next object that has to be reached by the SSc i (the object can be a target or the FS). A mission is every consecutive part of the sequence that start with a zero and ends with the following zero. Let s_k^i be the sequence of SSc i of mission k. It's length will be $n_{i,k} + 2$ due to the fact that a mission will always start and be completed in the FS, so $s_k^i(0) = 0$ and $s_k^i(n_{i,k} + 1) = 0$. For reference, in table 3.1, $s_1^1(2) = 2$, $s_2^1(2) = 0$ and $s_1^2(3) = 6$. In order to fill the empty part of the matrix, the number $J + 1$ is selected, like in position $(2,6)$ of table 3.1, but those numbers are not considered in the representation of s_k^i. To express all the possible configurations on the sequence s_k^i, a permutation of positions on mission k of SSc i $\sigma_i^k(\cdot) : \{0, 1, \ldots, n_{i,k}+1\} \rightarrow \{0, 1, \ldots, n_{i,k}+1\}$ is defined with the following constraints: $\sigma_i^k(0) = 0$ and $\sigma_i^k(n_{i,k}+1) = n_{i,k}+1$. To obtain the target on a specific position $p \in \{0, 1, \ldots, n_{i,k} + 1\}$, one may use $s_k^i(\sigma_k^i(p))$. To simplify the notation, let's define $\rho_k^i(\cdot) := (s_k^i \circ \sigma_k^i)(\cdot)$ as the function that, given a position on that specific mission, returns as output the space object on that position.

Table 3.1: Example of the form of the solution with six targets.

0	1	2	0	3	0
0	4	5	6	0	7

3.2.1 Objective function

The goal of the optimization is to minimize the total fuel consumption. Using eq. (2.10), the fuel consumption can be expressed using Δv of each maneuvers. The SSc might perform a planar change and a phasing maneuver before refueling the target. Let $m_{i,p}^k$ be the total mass of the SSc i at position $p = 0, \ldots, n_{i,k} + 1$ of the mission k. The total mass is the sum of the *dry mass* and the *fuel mass*. The dry mass, denoted as m_i^D, is the total mass of the SSc without the fuel mass (denoted as $m_{i,p}^{F,k}$). Due to assumption (A11), $m_{i,0}^{F,k}$ is always set to the fuel capacity of the SSc, denoted C_i from now on. Let $m_i = C_i + m_i^D$ be the total initial mass of the SSc, which is the initial mass at the beginning of every missions. Let $m_{i,j}^R$ the fuel that needs to be added to the target j. Consider the total mass of the SSc after reaching the first target $\rho_k^i(1)$ and after refueling it. Using the Rocket equation, the result will be the following:

$$m_i^D + m_{i,1}^{F,k} = \left(\left(m_i^D + C_i \right) \exp\left(-\frac{\Delta v_{0,\rho_k^i(1),1}}{g\, I_{sp}^i} \right) \right) \exp\left(-\frac{\Delta v_{0,\rho_k^i(1),2}}{g\, I_{sp}^i} \right) - m_{i,\rho_k^i(1)}^R$$

$$= \left(m_i^D + C_i \right) \exp\left(-\frac{\Delta v_{0,\rho_k^i(1),1} + \Delta v_{0,\rho_k^i(1),2}}{g\, I_{sp}^i} \right) - m_{i,\rho_k^i(1)}^R.$$

where $\Delta v_{0,\rho_k^i(1),1}$ and $\Delta v_{0,\rho_k^i(1),2}$ are the variation of velocity used from the FS to the first target of the sequence in the planar change and the phasing maneuvers, respectively. I_{sp}^i is the specific impulse of SSc i. To have a lighter notation, let's define $\Delta v_{j_1,j_2} = \Delta v_{j_1,j_2,1} + \Delta v_{j_1,j_2,2}$ as the total Δv used in the Rocket equation to reach target j_2 from j_1. The final fuel mass can be written as:

$$m_{i,1}^{F,k} = \left(m_i^D + C_i \right) \exp\left(-\frac{\Delta v_{0,\rho_k^i(1)}}{g\, I_{sp}^i} \right) - m_{i,j_1}^R - m_i^D$$

$$= C_i \exp\left(-\frac{\Delta v_{0,\rho_k^i(1)}}{g\, I_{sp}^i} \right) - m_i^D \left(1 - \exp\left(-\frac{\Delta v_{0,\rho_k^i(1)}}{g\, I_{sp}^i} \right) \right) - m_{i,j_1}^R.$$

Since the aim is to characterize the total mission consumption, let's consider p passages from the FS to target $\rho_k^i(p)$, which means $p < n_{i,k} + 1$. The following recursive equation can be written:

$$m_{i,p}^{F,k} = m_{i,p-1}^{F,k} \exp\left(-\frac{\Delta v_{\rho_k^i(p-1),\rho_k^i(p)}}{g\, I_{sp}^i} \right) - m_i^D \left(1 - \exp\left(-\frac{\Delta v_{\rho_k^i(p-1),\rho_k^i(p)}}{g\, I_{sp}^i} \right) \right) - m_{i,\rho_k^i(p)}^R.$$

The final result after substituting backwards will be the following:

$$m_{i,p}^{F,k} = C_i \cdot \exp\left(-\sum_{h=1}^p \frac{\Delta v_{\rho_k^i(h-1),\rho_k^i(h)}}{g\, I_{sp}^i} \right) - m_i^D \cdot \left(1 - \exp\left(-\sum_{h=1}^p \frac{\Delta v_{\rho_k^i(h-1),\rho_k^i(h)}}{g\, I_{sp}^i} \right) \right)$$

$$- \sum_{l=1}^p m_{i,\rho_k^i(l)}^R \exp\left(-\sum_{h=l+1}^p \frac{\Delta v_{\rho_k^i(h-1),\rho_k^i(h)}}{g\, I_{sp}^i} \right).$$

with the following artificial definition: $\sum_{h=p+1}^{p} \dfrac{\Delta v_{\rho_k^i(h-1),\rho_k^i(h)}}{g\,I_{sp}^i} = 0$ for every $p = 1,\ldots,n_{i,k}$. The index of the last sum starts with a +1 since the last action of the SSc is refueling when $p < n_{i,k}+1$ and therefore the last refueling quantity is not multiplied by any exponential. Finally, when $p = n_{i,k}+1$, the formula slightly changes:

$$
m_{i,n_{i,k}+1}^{F,k} = C_i \exp\left(-\sum_{h=1}^{n_{i,k}+1} \frac{\Delta v_{\rho_k^i(h-1),\rho_k^i(h)}}{g\,I_{sp}^i}\right) - m_i^D \left(1 - \exp\left(-\sum_{h=1}^{n_{i,k}+1} \frac{\Delta v_{\rho_k^i(h-1),\rho_k^i(h)}}{g\,I_{sp}^i}\right)\right)
$$
$$
- \sum_{l=1}^{n_{i,k}} m_{i,\rho_k^i(l)}^R \exp\left(-\sum_{h=l+1}^{n_{i,k}+1} \frac{\Delta v_{\rho_k^i(h-1),\rho_k^i(h)}}{g\,I_{sp}^i}\right).
$$

every p becomes $n_{i,k}+1$, with the exception of the penultimate summation, which does not have to count $m_{i,\rho_k^i(n_{i,k}+1)}^R$ because the SSc reaches the FS and it does not refuel any targets. That is coherent with the fact that in the formula, after the last refueling there are the last maneuvers to go back to the FS and thus just one exponential multiplying the last refueling. To obtain the total fuel used during during mission k of SSc i, the fuel at the end of the mission has to be subtract to the initial quantity in the following way:

$$
f_i^k = C_i - m_{i,n_{i,k}+1}^{F,k} =
$$
$$
= m_i\left(1 - \exp\left(-\sum_{h=1}^{n_{i,k}+1} \frac{\Delta v_{\rho_k^i(h-1),\rho_k^i(h)}}{g\,I_{sp}^i}\right)\right) + \sum_{l=1}^{n_{i,k}} m_{i,\rho_k^i(l)}^R \exp\left(-\sum_{h=l+1}^{n_{i,k}+1} \frac{\Delta v_{\rho_k^i(h-1),\rho_k^i(h)}}{g\,I_{sp}^i}\right)
$$

19

3.2.2 Mathematical Model

In table 3.2 are summarized all the variables of the model.

Table 3.2: Summary of model variables.

Var	*Description*
$\mathcal{I}$	Set of SSc.
$\mathcal{K}_i$	Set of missions of SSc i.
$\mathcal{J}$	Set of targets.
$\mathcal{J}^*$	Set of targets with the FS.
$\mathcal{J}_i^k$	Subset of targets associated to mission k of SSc i.
μ_i	Number of missions of SSc i.
$n_{i,k}$	Number of targets associated to mission k of SSc i.
$s_k^i(\cdot)$	Sequence of mission k of SSc i, it contains initial and final zeros.
$\sigma_k^i(\cdot)$	Permutation of positions of mission k of SSc i.
$\rho_k^i(\cdot)$	Function that returns the space object on a given position in mission k of SSc i.
m_i^D	Dry mass of the SSc i.
$m_{i,p}^{F,k}$	Fuel mass of SSc i after refueling the target in position p of mission k.
m_i	Total mass of the SSc i with the full fuel tank.
$m_{i,j}^R$	Refueling mass requested from target j.
C_i	Total capacity of the SSc i.
$\Delta v_{j_1,j_2}$	Variation of velocity applied to pass from j_1 to j_2.
I_{sp}^i	Specific impulse of SSc i.
$\varepsilon_{i,p}^k$	Variable used to consider a fuel buffer in the fuel constraints.

Let's consider the following model:

$$\min_{\mathcal{J}_i^k, \sigma_i^k} \quad \sum_{i=1}^{I} \sum_{k=1}^{\mu_i} \left[m_i \left(1 - \exp\left(- \sum_{h=1}^{n_{i,k}+1} \frac{\Delta v_{\rho_k^i(h-1), \rho_k^i(h)}}{g\, I_{sp}^i} \right) \right) \right.$$
$$\left. + \sum_{l=1}^{n_{i,k}} m_{i,\rho_k^i(l)}^{R} \, \exp\left(- \sum_{h=l+1}^{n_{i,k}+1} \frac{\Delta v_{\rho_k^i(h-1), \rho_k^i(h)}}{g\, I_{sp}^i} \right) \right] \tag{3.1}$$

$$\text{s.t.} \quad \bigcup_{k \in \mathcal{K}} \bigcup_{i=1}^{J_k} \mathcal{J}_i^k = \mathcal{J}, \tag{3.2}$$

$$\mathcal{J}_i^{k_1} \cap \mathcal{J}_i^{k_2} = \emptyset, \qquad \forall\, i \in \mathcal{I},\ \forall\, k_1 \neq k_2 \in \mathcal{K}_i, \tag{3.3}$$

$$\mathcal{J}_{i_1}^{k_1} \cap \mathcal{J}_{i_2}^{k_2} = \emptyset, \qquad \forall\, i_1 \neq i_2 \in \mathcal{I},\ \forall\, k_1 \in \mathcal{K}_{i_1},\ k_2 \in \mathcal{K}_{i_2}, \tag{3.4}$$

$$m_{i,p}^{F,k} \geq m_{i,p+1}^{F,k} + \varepsilon_{i,p}^{k} \qquad \forall\, i \in \mathcal{I},\ k \in \mathcal{K}_i, \forall\, p = 0, \dots, n_{i,k}. \tag{3.5}$$

$$m_{i,n_{i,k}+1}^{F,k} \geq 0, \qquad \forall\, i \in \mathcal{I},\ k \in \mathcal{K}_i. \tag{3.6}$$

As already stated before, 3.1 is the objective function. The minimization is performed over the possible sets for every mission and the permutation for every mission. The sequence s_k^i is not a decision variable, since it can uniquely be determined by the previous two quantities. Constrain 3.2 assures that the union of every mission subset forms the entire target set. Constrain 3.3 enforced every mission set of the same SSc to be disjoint, while constrain 3.4 enforced the same condition between every other missions on different SScs. Constrains 3.5 and 3.6 are feasibility constraints: they make sure every maneuvers are feasible by imposing a continuous decreasing of the fuel mass when the position increases. $\varepsilon_{i,p}^k$ is a decision variable used to relax the constraints: the total fuel of the SSc cannot be zero when $p < n_{i,k} + 1$ because otherwise the following maneuver would be infeasible. Therefore $\varepsilon_{i,p}^k$ represents the fuel buffer that the SSc needs to keep to performs the following maneuvers. in eq. (3.6) there is not such variable since the mission ends.

Chapter 4

Optimization framework

Before explaining the optimization process, the definition from [21] of Neighborhood search is given. Consider an instance of combinatorial optimization problem I. Let $X(I)$ be the set of feasible solutions of the considered instance (called just X to be short). Let also $c : X \to \mathbb{R}$ be a *cost function* that associates a cost to a feasible solution. A *Neighborhood* is a function $N : X \to X$ that, for a solution $x \in X$, returns a set of solutions $N(x) \subseteq X$. This definition suggests a definition of *local optimality* with respect a neighborhood, in fact the condition of a solution $x \in X$ to be locally optimal in this sense is $c(x) \leq c(x'), \forall x' \in N(x)$. In this setting, a *Neighborhood search algorithm* can be constructed as follows. Let x be an initial solution, by iteratively computing $x' = \arg\min_{x'' \in N(x)} c(x'')$ and updating $x = x'$ if $c(x') \leq c(x)$, the solution can be improved until it reaches a local optimal with respect the chosen neighborhood. Typically, the neighborhood needs to be explicitly defined to construct such algorithms, but some of them can implicitly define the neighborhood. The *LNS* metaheuristic proposed by [22] is an example of implicitly definition of neighborhood by using some methods called *destroy* and *repair*. More detail about destroy and repair methods are shown in Chapter 5. This kind of search particularly find applications in tightly constrained settings, since the idea of a large neighborhood helps the heuristic to explore the solution space easily. Algorithm 1 describes the LNS metaheuristic. The algorithm uses three principal variables: the overall best solution x^b, the current solution x^c and the temporary solution x^t. A temporary solution is obtained from the current solution using the destroy and repair methods and it is evaluated. If a certain acceptance criterion is met, then the current solution is update with the temporary one. More detail on this criterion will be explained in the Section 4.2. If the temporary solution is better than the best one, then it become the new optimal solution. It is worth to notice that the LNS does not search the entire neighborhood of solutions, but it sample the neighborhood with the destroy and repair methods to heuristically find a good solution.

Algorithm 1 Large Neighborhood Search (LNS) [21]

Input: Feasible solution x^c.
Output: Best solution found x^b.
$x^b \leftarrow x^c$.
repeat
  $\quad x^t \leftarrow r(d(x^c))$.
  $\quad$**if** $accept(x^t, x^c)$. **then**
  $\quad\quad\llcorner\ x^c \leftarrow x^t$.
  $\quad$**if** $c(x^t) < c(x^b)$ **then**
  $\quad\quad\llcorner\ x^b \leftarrow x^t$.
until *stopping criterion is met*;
return x^b

4.1 Adaptive Large Neighborhood Search

ALNS was proposed by [23] and extends the LNS framework by allowing the usage of more destroy and repair methods. Algorithm 2 has basically the same framework as algorithm 1, a new solution is constantly proposed and checked. The main difference is that, since there are a set of destroy operators Ω^d and repair operators Ω^r, in every iteration a single destroy and and a single repair are chosen. A weight is associated to every destroy and repair operators, those weights are dynamically updated to adapt better to the specific instance and they are used to calculate the probabilities of drawing the destroy and repair method they are associated with. This updating method also provides a way to continuously use good operators and evaluates the contribution of the operator to the optimization process.

Algorithm 2 Adaptive Large Neighborhood Search (ALNS) [21]

Input: Feasible solution x^c
Output: Best solution found x^b
$x^b \leftarrow x^c \ \ w^d \leftarrow (1, \ldots, 1), \ \ w^r \leftarrow (1, \ldots, 1)$.
repeat
  $\quad$Select destroy and repair methods $d \in \Omega^d$ and $r \in \Omega^r$ using w^d and w^r.
  $\quad x^t \leftarrow r(d(x^c))$.
  $\quad$**if** $accept(x^t, x^c)$ **then**
  $\quad\quad\llcorner\ x^c \leftarrow x^t$.
  $\quad$**if** $c(x^t) < c(x^b)$ **then**
  $\quad\quad\llcorner\ x^b \leftarrow x^t$.
  $\quad$Update w^d and w^r.
until *stopping criterion is met*;
return x^b

4.1.1 Weight's update

Consider a destroy and repair set (Ω^d and Ω^r). Let $nd = |\Omega^d|$ and $nr = |\Omega^r|$, moreover consider the weights associated to the destroy and repair methods, respectively $w^d \in \mathbb{R}^{nd}$ and $w^r \in \mathbb{R}^{nr}$. Initially, all weights are typically set to one. The probability associated to every methods are computed at each iteration in the following way:

$$\pi_j^d = \frac{w_j^d}{\sum_{k=1}^{nd} w_k^d}, \qquad \pi_j^r = \frac{w_j^r}{\sum_{k=1}^{nr} w_k^r},$$

where j indicates the j-th component of the considered vector. After extracting the destroy and repair, the solution is calculated and it is compared to the current and best solution as suggested in algorithm 2. When a destroy and repair is used, a score ψ is determined in the following way:

$$\psi = max\Big\{ \ \psi_1, \ \psi_2, \ \psi_3, \ \psi_4 \ \Big\},$$

where

$$
\begin{cases}
\psi_1 = \delta_1 \cdot (c(x^t) < c(x^b)), & x^t \text{ is the new best one} \\
\psi_2 = \delta_2 \cdot (c(x^t) < c(x^c)), & x^t \text{ is better than the current one} \\
\psi_3 = \delta_3 \cdot (\text{accept}(x^t, x^c)), & x^t \text{ is accepted} \\
\psi_4 = \delta_4 \cdot (1 - \text{accept}(x^t, x^c)), & x^t \text{ is rejected}
\end{cases}
$$

All the conditions in parentheses return 1 if the condition is true or 0 if it is false. $\delta_1, \delta_2, \delta_3, \delta_4$ are parameters used to assign the successful score to the operators. Normally $\delta_1 \geq \delta_2 \geq \delta_3 \geq \delta_4 \geq 0$, the goal is to reward the operators that provides value to the solution ($\delta_1, \delta_2, \delta_3$) and discourage operators that reject solutions (δ_4), since a rejected solution is a wasted iteration. Let jd and jr be the indices of the destroy and repair operators used in the iteration respectively. The corresponding weight component is updated in the following way:

$$w_{jd}^d = \lambda w_{jd}^d + (1 - \lambda)\psi, \qquad w_{jr}^r = \lambda w_{jr}^r + (1 - \lambda)\psi,$$

where $\lambda \in [0,1]$ is the decay parameter that express how much important is the new increment and the old value with a convex combination. ALNS heuristic typically favors complex repair methods, since they usually give high quality solutions. Thus, this weight's update does not consider how much time-consuming are the methods. To obtain a trade-off between time consumption and solution quantity, one may normalize ψ with a measure of the time consumption.

4.2 Acceptance Criteria

The acceptance criterion is typically a function that allows a change in the current solution, which means a different starting points for the next destroy and repair cycle. Some examples are provided by [24] in chapter iii. Some simple choice may be accepting everything (Random Walk) or accepting every new solutions that are better than the best so far (Greedy Acceptance). Threshold Accepting discard every solutions that are below a certain fixed or variable threshold, while *Simulated Annealing (SA)* accepts every better solutions and, with a certain probability, accept some worst solution. The key principle is that even a worse solution may led to some better solutions, so it has a higher exploration grade than the greedy acceptance. A worse solution is accepted with probability $\exp((c(x^c) - c(x^t))/T)$, where $T > 0$ is called *temperature*. The main goal of the temperature is to vary the acceptance probability so that it allows more exploration at the beginning of the search process and to exploit more during the final part of the process. The temperature is initialized at $T_0 > 0$ and decreases gradually through a decreasing parameter α at each iteration: $T_{new} = \alpha T_{old}$, where $0 < \alpha < 1$. $c(x^c) - c(x^t)$ is a term used to take into account the goodness of the solution: the more x^t is closed to x^c in terms of cost, the higher the acceptance probability will be. Since the probability is used when x^t is worse than x^c, it means that $c(x^c) - c(x^t) < 0$ so the probability will be between zero and one. In Section 6.1, Greedy Acceptance and SA are compared.

Chapter 5

ALNS Operators

In this chapter, the core idea of implicitly defining the neighborhood is expanded, describing some general aspects of destroy and repair methods. A destroy method takes a solution as input and returned an incomplete solution and it's destroyed part as an output. This concept is illustrated in fig. 5.1. The initial solution (left

Figure 5.1: An application of a destroy method (center) and a repair method (right) of an initial solution (left).
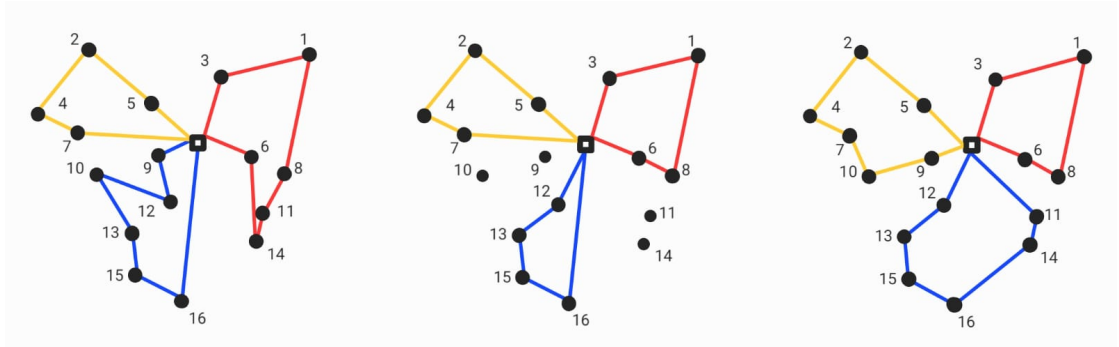
of fig. 5.1) is composed by three tours: tour yellow (Y) composed by [5, 2, 4, 7], tour red (R) composed by [6, 14, 11, 8, 1, 3] and tour blue (B) composed by [9, 12, 10, 13, 15, 16]. After a destroy method is applied, the configuration in the center of fig. 5.1 is obtain, which corresponds to this incomplete solution : $Y = [5, 2, 4, 7]$, $R = [6, 8, 1, 3]$, $B = [12, 13, 15, 16]$ with the set of destroyed node, that is [9, 10, 11, 14]. The selection of those destroyed node is what characterize the destroyer and, consequently, the neighborhood. The repair method will take the output of the destroyer as input and it will rebuild a new solution out of it. The right of fig. 5.1 shows the following new constructed solution: $Y = [5, 2, 4, 7, 10, 9]$, $R = [6, 8, 1, 3]$, $B = [12, 13, 15, 16, 14, 11]$.

Alternately, the destroy and repair methods can be seen as *fix* and *optimize* operators, respectively. In particular, the destroy fixes some part of the solution, while the optimize method attempts to improve the solution using what has been removed. A similar framework called *ruin and recreate* was proposed by [24].

5.1 Destroy Operators

Typically, all destroy methods includes some stochasticity, because the main goal is exploring a large neighborhood and thus different part of the solution need to be placed by the repair after the destruction. The most crucial part of a destroy method is the *degree of destruction.* Ideally, one may want a high grade of destruction to explore a large set of solutions, but if it is too large, there is a higher chance to re-optimization that throws away important computational resources. Moreover, since a destroy method needs to reach a good portion of the search space, it should not focus on destroying always a specific part of the solution. In Section 6.3 some destroy policies are discussed and compared. Two important characteristic of destroy methods are *diversification* and *intensification.* Stochasticity is usually employed to diversify the search, while intensification is choosing the worst part of the solution so that the repair may re-insert them in a better place. This work proposed different kind of destroy methods and compared them in Section 6.4.

- *Destroy First*: It destroys the first target of one or more tours.

- *Destroy Last*: It destroys the last target of one or more tours.

- *Destroy Random*: It destroys randomly some targets.

- *Destroy Tour Cost*: It destroys some tours starting from the most expensive ones.

- *Destroy Tour Small*: It destroys some tours starting from the smallest ones.

- *Destroy Tour Random*: It destroys some tours randomly.

- *Destroy SSc Random*: It destroys targets from one or more SSc randomly.

- *Destroy SSc Cost*: It destroys targets from one or more SSc starting from the most expensive one.

- *Destroy Related Greedy*: It destroys some similar targets.

- *Destroy Related Random* from [3]: It destroys some similar targets. It may not chooses the most similar targets, a level of stochasticity is introduced to avoid always picking the same targets.

The following relatedness measure $R(i,j)$ from [3] has been used to implement some *Related* destroyer. This measure evaluates the similarity of two targets. A high value correspond to high relatedness of the two involved targets. The measure is computed as follows:

$$R(i,j) = \frac{1}{C'_{i,j} + V_{i,j}}.$$

$V_{i,j}$ is equal to one if target i is refueled by the same SSc of j and it is zero otherwise. $C'_{i,j}$ is a normalized approximation of the cost of getting from target i to j:

$$C'_{i,j} = \frac{C_{i,j}}{\max(C)}.$$

$C_{i,j}$ represents the approximation of reaching j from i. In general denotes the distance between two nodes in a classical VRP problem ([22]). The approximation is the same used in [3]:

$$C_{i,j} = \beta \ |\theta_{pc}^{(i,j)}| + (1 - \beta) \ \theta_{i,j},$$

where $\theta_{pc}^{(i,j)}$ is the dihedral angle of the two target's orbit [eq. (2.11)] and $\theta_{i,j}$ is their phasing angle [eq. (2.15)]. $\max(C)$ is the normalizing factor, the maximum entry of the whole matrix C. The Destroy Related Random and the Destroy Related greedy both start with extracting the first target to remove and by iteratively computing the relatedness measures. The main difference is that the Related Greedy continuously considers the most similar target and removes it, while the Related Random considers a parameter $p \geq 1$ used to include randomness and not selecting always the first option by choosing the $\lfloor r^p \rfloor$-th element, where r is a random number between zero and one.

5.2   Repair operators

The repair method rebuild the solution from the output of the destroy method. When designing a repair method, one big decision is whether construct the best solution or heuristically find a good one. An optimal repair may led to high quality solutions, but it is computationally heavy and has difficulty to leave local minima, unless a big part of the solution is destroyed. On the other hand, heuristic repair are faster but it may take more iteration to get a high quality solution. This work proposes two repair methods and adopted a third repair method from [3]. The repairers are designed to return a feasible solution, so various check with the simulator are iteratively performed to ensure all rendezvous are feasible, this is the reason why assumption (A15) is required. Firstly, a *Random Repair* has been implemented. It considers the first mission, tries to insert some random targets in

random positions for a certain portion of the destroyed set. If the resulting mission is feasible, the mission is updated and the target is removed from the destroyed set. If all the resulting missions are infeasible, this process is repeated for the next tour available. All tours are potentially touched, the algorithm starts by the first mission of the first SSc, check all the first tours, then check all the others. If some destroyed targets remains after checking all missions, they will be added to a new mission after refueling at the FS. This repair methods aims to be computationally cheaper than the other, trying to obtain a more compact solution in a fastest time, based on the following fact stated in [13]: the more targets are refueled in the same mission, the less fuel will be used during the maneuvers. Algorithm 3 presents a pseudo-code on the Repair Random procedure. The repair method from [3], called in this work *Repair Insertion Simulation* from now on, is based on the *farthest insertion heuristic*. The goal is to obtain a high quality solution, checking all the possible position where a target can be inserted. The order of insertion is determined by the biggest "insertion cost". Let $\Delta f_{j,i,p}$ be the objective value changes caused by inserting the removed target j in position p of the SSc i. The insertion cost has been calculated in the following way:

$$c_j = \min_{p,i} \Delta f_{j,i,p},$$

in other words, the insertion cost is the minimum variation of objective function obtained by adding a target j into the sequence. The farthest insertion heuristic aims to firstly place the most difficult targets, which are the targets with higher insertion cost, to reduce difficulties of placing those targets in the end of the insert procedure due to feasibility constraints. The algorithm starts with creating a structure used to simulate all the possible destroy targets in all possible positions. Then the chosen target is computed in the following way:

$$j^* = \arg\max_j \min_{p,i} \Delta f_{j,i,p},$$

and is inserted in the position and SSc associated to the insertion costs. Then the algorithm updates the structure used to simulate and calculate again j^* until it ends all destroyed targets. A similar repair called *Insertion Related* based on the farthest insertion heuristic is proposed to try to consider both high quality solutions and computational speed. This method tries to insert the destroyed target near it's most similar target. The structure is basically the same as the Insertion Simulation, but it aims to be lighter because it does not simulate every targets in every positions, thus it computes the relatedness measure between all destroyed targets and all fixed target. The destroyed target which is the least similar to all the destroyed set is chosen, then placed near the most similar to him. Only then, the simulation assures feasibility, if the resulting sequence turns out to be

infeasible, the second most similar target is chosen and the feasibility is checked again. Algorithm 4 shows the Repair Insert Simulation procedure from [3].

Algorithm 3 Repair Random Procedure

Input: Set of destroyed target: x^d, Information about the partial solution.
Output: Constructed tours for all SScs.
Initialize current tour and current SSc.
Initialize Simulation State for every SSc.
repeat
    if *current tour* $= \emptyset$ **then**
        Choose randomly a target and a position.
        Update the new tour.
        Remove target from x^d.
    else
        $x^r \leftarrow x^d$;
        $nInfeas \leftarrow 0$.
        repeat
            Reset Simulation State of the current SSc.
            Choose randomly a target and a position.
            Create the new tour and simulate it.
            if *tour is feasible* **then**
                Save new tour.
                Remove target from x^d.
            else
                $nInfeas \leftarrow nInfeas + 1$.
            Remove target from x^r.
        until $nInfeas > nSearch$ or $x^r = \emptyset$;
        if $nInfeas == nSearch$ or $x^r = \emptyset$ **then**
            Update Simulation State of current SSc for a possible following tour.
            Switch to another SSc and/or tour.
until $x^d = \emptyset$;

Algorithm 4 Repair Insert Simulation Procedure [3]

Input: Removal set: x^r, destroyed set: x^d.
Output: Repaired set: x^t.
Initialize repaired set $x^t \leftarrow x^d$.
repeat
    for t_i^r in x^r **do**
        Calculate $\Delta f_{i,k}$ of t_i^r based on x^t.
        Find the minimum cost position $P_{i,k}$ of t_i^r in x^t.
    Find t_i^r in x^r with maximum $\Delta f_{i,k}$.
    Insert t_i^r into $P_{i,k}$ in x^t.
    Remove t_i^r from x^r.
until $x^r == \emptyset$;
return x^t

Chapter 6

Computational Results

In this chapter, some experiments are performed to test various strategies on a real-world orbital element of a fixed scenario. Fourteen launched Chinese geosynchronous satellites taken from [3] have been chosen, along with the FS, which has been set to a similar orbit. Every SScs are in the FS ready to be launched. Table 6.1 shows all orbital elements involved. [3] does not specify their mass, since they addressed the

Table 6.1: Orbital elements of a real-world scenario, from [3]

ID	Name	Inclination	RAAN	True Anomaly
0	FS	2.00°	60.00°	0.00°
1	Beidou2_G7	1.60°	66.76°	278.27°
2	Beidou2_G8	0.30°	328.08°	156.03°
3	Beidou_G1	1.80°	45.11°	252.16°
4	Beidou_G2	7.77°	52.63°	328.00°
5	Beidou_G3	1.89°	52.10°	274.21°
6	Beidou_G4	1.06°	59.65°	144.68°
7	Beidou_G5	1.45°	67.40°	288.52°
8	Beidou_G6	1.86°	85.65°	319.30°
9	Chinasat_11	0.09°	103.25°	331.94°
10	Fengyun_2E	5.00°	68.04°	285.07°
11	Fengyun_2F	2.80°	83.11°	224.48°
12	Tianlian1_01	4.81°	71.74°	337.75°
13	Tianlian1_02	2.21°	74.98°	229.24°
14	Tianlian1_03	0.99°	98.18°	230.86°

repairing problem, thus the mass data of the targets are set as indicated in table 6.2. SSc's mass and parameters (table 6.3) are instead taken from [13]. Every SScs

Table 6.2: Mass table

Object	m^D [Kg]	m^{TOT} [Kg]	m^R [Kg]
SScs	500	5000	-
Targets	3000	3700	400

refuel the targets by one Kg every two minutes, while the FS refuels the SScs one Kg per minute. Every experiment has been run five times to see how the stochastic

Table 6.3: Vehicles parameters

Parameter	*Value*
Specific Impulse [s^{-1}]	0.3058
Refueling speed SSc [Kg/s]	0.0083
Refueling speed FS [Kg/s]	0.0166

aspect of the framework influenced the process. Every experiments are conducted with the random seed set to "12345" for replicability. Moreover, every experiments have been conducted with the following characteristics, except when explicitly stated otherwise: all destroyers and repairs are used, a fixed degree of destruction of 30%, as done previously in [3], SA acceptance criterion is adopted. Moreover the initial solution adopted is the simple solution one may think: every mission will have just one single target. This solution is surely feasible due to assumption (A14). The work is organized as follows. Section 6.1 evaluates some parameters testing the best configuration among them and confronts the SA acceptance criterion with the Greedy approach. Section 6.2 runs simulations starting by different initial solutions, build using the previous described repair methods and studies how solutions change. Section 6.3 applies different destruction policies based on changing the degree of destruction. Section 6.4 compares the weights and results obtained by removing some destroy method to address the contribution and the importance of each destroyers and repairs. When a configuration has proven to bring better results, it is adopted for the following experiments. The complete results are stored in the "Results" folder of the git Hub project, for more information see appendix A.

6.1   Parameters Tuning and Acceptance criteria

In the section, the δ and λ parameters from Subsection 4.1.1 along with α and $T0$ from Section 4.2 are tuned. The configurations chosen for the deltas are shown in table 6.4.   Those values have been chosen to test how the different scale modifies

<table>
<tr><td colspan="5">**Table 6.4:** Delta parameters</td><td colspan="3">**Table 6.5:** SA parameters</td></tr>
<tr><td>*Configuration*</td><td>$\boldsymbol{\delta_1}$</td><td>$\boldsymbol{\delta_2}$</td><td>$\boldsymbol{\delta_3}$</td><td>$\boldsymbol{\delta_4}$</td><td>*Configuration*</td><td>$\boldsymbol{\alpha}$</td><td>$\boldsymbol{T0}$</td></tr>
<tr><td>1</td><td>10</td><td>7</td><td>3</td><td>1</td><td>0</td><td>0.995</td><td>400</td></tr>
<tr><td>2</td><td>2</td><td>1.5</td><td>1</td><td>0.5</td><td>1</td><td>0.9</td><td>250</td></tr>
<tr><td>3</td><td>1</td><td>0.7</td><td>0.3</td><td>0.1</td><td>2</td><td>0.8</td><td>100</td></tr>
</table>

the optimal value and the weights of the operators. The values chosen for the decay parameter are $1 : 0.25$, $2 : 0.5$, $3 : 0.75$ to test how much the previous weight is important to determine the algorithm's optimal value. In table 6.6 are summarized the results over all replicas. The most used destroyers are the Random Destroyer, the Related Random destroyer and the First Destroyer, however some especially good solutions are obtained with the Destroy Last. The Repair Insert Simulation is widely used, scoring sometimes double the amount of weights of other repairers. The change of deltas and lambda does not seems to modify to much the behavior of the method. The scale of deltas does not affect the number of times a Destroy or repair is chosen. The decay method provides more uniform picking when it is low, but still does not change the solution dramatically. The configuration that has been adopted from now on is (2,1). The next parameters to tune are the the one from the SA acceptance criterion. After empirically check how the temperature decays, the parameters in table 6.5 have been chosen. Configuration zero has already been run during the tuning of deltas and lambda. First of all, $T0$ has been set to 400 and the tuning on configuration one and two of α is performed. After taking the best value for α, configuration one and two of $T0$ have been tested. The new proposed parameters for the SA criterion are thought to see if more exploitation at the end of the simulation could improve the solution.

Figure 6.1: Value of the current solution when $\alpha = 0.995$
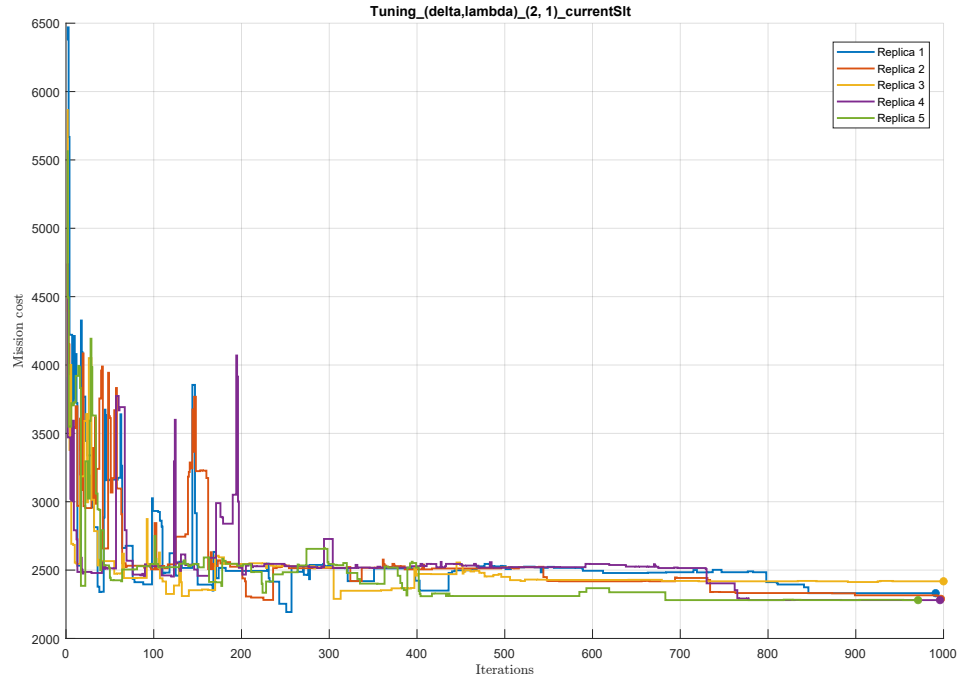
Figure 6.2: Value of the current solution when $\alpha = 0.8$
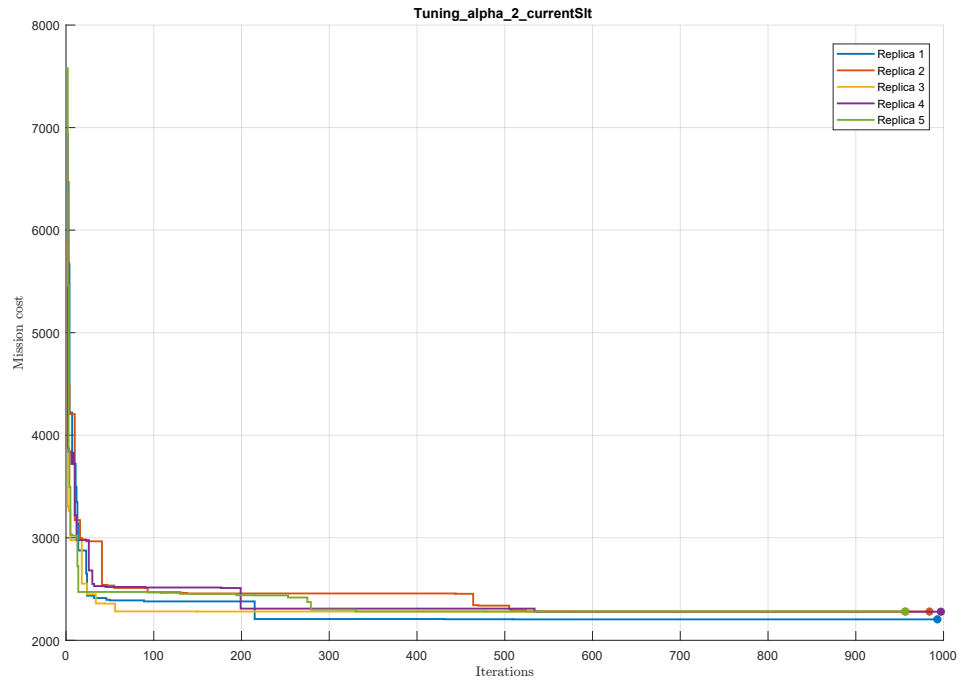
In fig. 6.1 and fig. 6.2 the value of the current solution is reported with two different values of alpha. It clearly shows how setting $\alpha = 0.8$ completely cancels the exploration part. Since the configuration with alpha equal to 0.9 returns an iteration with fuel consumption equal to the best found so far and less fuel consumption in other iterations, that value will be used for now on. Regarding $T0$, a similar reasoning can be apply, thus a new best solution of 2186.8 has been found. Another simulation has also been performed with the greedy acceptance criterion, this approach led to constant weights, since all the progress were done in a relatively small number of iterations with respect to the total number of iteration. The results shows a minima of 2197.9 Kg.

Table 6.6: Tuning and Accept Results

Configuration	Rep 1	Rep 2	Rep 3	Rep 4	Rep 5	Sum
(1,1)	2302.7	2280.5	2279.1	2280.5	2280.5	11423.3
(1,2)	2209.7	2280.5	2279.1	2206.7	2280.4	11256.4
(1,3)	2280.5	2303.1	2280.4	2280.5	2280.4	11425.0
(2,1)	2193.1	2281.8	2289.5	2280.5	2280.5	11325.4
(2,2)	2280.5	2280.5	2280.5	2280.5	2280.5	11402.5
(2,3)	2280.5	2279.1	2280.5	2280.5	2280.4	11401.0
(3,1)	2280.4	2280.5	2280.5	2280.4	2280.5	11402.3
(3,2)	2280.5	2297.2	2279.1	2280.5	2302.8	11440.1
(3,3)	2280.4	2280.4	2280.4	2280.5	2280.4	11402.1
alpha1	2204.4	2279.1	2193.1	2280.5	2280.5	11237.6
alpha2	2204.4	2280.5	2280.5	2279.1	2280.5	11325.0
T01	2227.3	2197.9	2280.5	2186.8	2280.5	11173.0
T02	2204.4	2280.5	2279.1	2313.7	2279.1	11356.8
Accept	2280.5	2280.5	2331.2	2197.9	2279.1	11369.2

6.2   Initial Solution

In this section, two new initial solutions are constructed using the Repair Random and the Repair Insertion Simulation. Moreover, the solution from table 6.8 has been put as initial solution to see if can be improved more. The idea is to begin with a better quality solution instead of an easy one to compute. The solution in table 6.7 presents a total fuel consumption of 4140.6 Kg, while the solution from table 6.9 uses 3854.7 Kg.

Table 6.7:
Solution from the Random Repair

Object	Sequence
SSc 1	$5,3,14 - 9,4,6 - 8,13$
SSc 2	$11,10,1 - 2,12,7$

Table 6.8:
Best solution obtained from Tuning

Object	Sequence
SSc 1	$9,8,4 - 7,10,1,14$
SSc 2	$12,11,13,2 - 5,3,6$

Table 6.9: Solution from the Insertion Simulation Repair

Object	Sequence
SSc 1	$8,12,6 - 14,13,11 - 5,14,7 - 1,10$
SSc 2	$9,3,2$

In table 6.10 are shown the results for every initial solutions. The minimal value found so far has been reached by starting with the Repair Random, since the other replica returns lower values, from now on that solution has been set as the new initial solution. This shows how a good balance of exploitation and exploration can be the perfect recipe to get a better solution. Starting by a better solution can improve the solution, but starting from a solution that is too good, like the old best solution, can lead to local minima or can lead to no improvement at all.

Table 6.10: Initial Solution Results

Configuration	Rep 1	Rep 2	Rep 3	Rep 4	Rep 5	Sum
Random	2280.5	2280.4	2207.4	2280.5	2186.8	11235.6
Simulation	2280.5	2280.5	2336.8	2280.5	2280.5	11459.0
Best	2193.1	2193.1	2193.1	2193.1	2193.1	10965.5

6.3 Destruction Policy

An alternative strategy is changing the degree of destruction during the optimization. [22] suggested to increase the degree of destruction during the optimization, to better avoid local minima. This policy (called in this work "Increasing" for short) is compared with the fixed policy used before and a Random policy, which changes randomly the degree of destruction at every iteration. In table 6.11 a summary of the results are presented. They both present higher fuel consumption with respect the fixed policy. More or less after two hundred iterations, the increasing policy does not update any more the current solution. After a while, the temperature

of the SA makes harder for a worse solution to be accepted, moreover the total destruction of the solution causes a lack of exploitation, which provides difficulties to get a better solution.

Table 6.11: Destroy Policy Results

Configuration	Rep 1	Rep 2	Rep 3	Rep 4	Rep 5	Sum
Random	2406.6	2209.7	2337.9	2279.1	2302.7	11535.9
Increasing	2280.5	2281.8	2280.5	2410.8	2302.8	11556.4

6.4 Operators Comparison

Finally, the following comparison between opearators is proposed. For Destroy methods five simulation, summarized in table 6.12 has been run. Every configuration is obtained by removing some destroyers, in table 6.13 are shown the results. The importance of each destroy methods can be seen by the final weights and how removing some destroy affects the performance. The first observation is that the values still does not differs too much: there are some higher and lower values, but generally they do not differs too much between the results previously discussed. Secondly, the SSc destroyer generally have very low weights. The most high values of weights are reached by the First Destroyer, Related Random destroyer, Tour and Small destroyer and Last Destroyer. That happens because in the first part of the optimization, it is surely convenient to try to populate the missions with more targets possible. Additionally, the random versions are typically more used, also the cost versions provides some importance. For the Repair methods, six simulations

Table 6.12: Configurations of Destroyed Set chosen to compare

Configuration	Removed Destroyer
1	Random and Related
2	First and Last
3	all involved with missions
4	all involved with SScs
5	Configuration 2, 3 and 4 together

in table 6.14 have been performed instead. In table 6.15 are presented the results for every configuration. It can be observed that the last three configurations have typically higher values than the first three. This shows the strength of the ALNS

Table 6.13: Destroy Comparison Results

Configuration	Rep 1	Rep 2	Rep 3	Rep 4	Rep 5	Sum
1	2279.1	2280.5	2280.5	2280.5	2207.4	11328.0
2	2193.1	2279.1	2280.5	2279.1	2280.5	11312.3
3	2279.1	2297.5	2280.5	2279.1	2280.5	11416.7
4	2331.2	2280.5	2197.9	2280.5	2280.5	11370.6
5	2279.1	2280.5	2280.5	2280.5	2207.4	11328.0

over the LNS: more operators means enlarge the neighborhood and may decrease the optimal value. The configurations with lower fuel cost are the one without the repair Insert Simulator proposed by [3]. This is not surprising since the goal of the heuristic is exactly trying to insert the targets in the best way possible, moreover the simulation provides a sure way to decrease the objective function. The random repair provides better results than the Insertion Similarity. This may happens because the random repair is build with the goal of saturate missions, while the Related Insert may divides and adds missions since it just considers the relatedness, potentially increasing the total fuel cost. The fact that qualitative repairers are more rewarded shows how important is the construction of a stable and controlled optimization.

Table 6.14: Configurations of Repair Set chosen to compare

Configuration	Removed Repair
1	Random
2	Insertion Simulation
3	Insertion Relatedness
4	Configuration 1 and 2 together
5	Configuration 2 and 3 together
5	Configuration 1 and 3 together

6.5 Best Solution obtain by Algorithm

In this section, some final observations on the best solution are reported in table 6.8. The most frequent solutions ranges between 2279.1 to 2280.5 Kg. One interesting fact is that most of those solutions are just mission permutation of the same solutions, which means that is not important where tours are placed in the sequence since

Table 6.15: Repair Comparison Results

Configuration	Rep 1	Rep 2	Rep 3	Rep 4	Rep 5	Sum
1	2279.1	2280.5	2280.5	2331.2	2280.5	11451.8
2	2361.3	2338.4	2316.2	2309.2	2279.1	11504.2
3	2190.8	2279.1	2279.1	2279.1	2280.5	11308.6
4	2761.3	2436.0	2419.9	2410.6	2338.5	12366.3
5	2207.4	2347.3	2392.9	2297.2	2296.5	11541.3
6	2331.2	2418.0	2279.1	2279.1	2280.5	11587.9

most of the simulations returns similar total fuel consumptions. The minimum obtained by the algorithm is 2186.8 Kg. The most common solution has a total fuel consumption of 2280.5 Kg and it is composed by the following tours: t_1 : $[0, 11, 13, 2, 0]$, t_2 : $[0, 8, 1, 14, 5, 0]$, t_3 : $[0, 12, 4, 7, 10, 0]$, t_4 : $[0, 9, 3, 6, 0]$. A slightly better solution is obtained by replacing t_2 and t_3 with t_5 : $[0, 8, 4, 7, 10, 0]$ and t_6 : $[0, 12, 1, 14, 5, 0]$. The best solution found has been obtained starting by the solution in table 6.7, using every destroy and repair, using configuration (2,1) for delta and lambda and by choosing SA with $\alpha = 0.9$ and $T0 = 400$. It is interesting to observe that part of the previous tours where embedded in some of the new tours, for instance the tour $[0, 7, 10, 1, 14]$ is composed by the last two targets of t_3 and the two middle target of t_2 and t_6. This suggest that some portion of the tours can be destroyed and recomposed to create better solutions. Thus, this particular strategy is effective also due to the structure of the considered scenario: a tour could not have more than four targets, because the fuel tank of the SSc and the requests from the targets impose that. To improve a good solution and to be sure not to ruin it, at least that number of targets to every mission have to be maintained.

Table 6.16: Best solution obtained from Tuning

Object	Sequence
SSc 1	$7,10,1,14 - 13,3,6 - 12,5,11,2 - 9,8,4$

Chapter 7

Conclusions

This work provides an example of both simulation and optimization for OOS. It addresses some way to improve the solution and it remarks the importance of balancing exploration and exploitation during the optimization process. It presents how problem's characterization may lead to discover new patterns and suggest some innovations. It shows how hard is improving the solution when too much randomness are present, the limits of exploiting good solutions too strictly and how a worse solution can lead to the best one found. But more importantly, it remarks how different approaches and way of thinking provides a better view of the problem and a better final result. Still, a lot of future improvement can be achieved. More constraints can be added: a maximum amount of variation of velocity to better preserve the satellite as done by most of the literature cited; deadline constrains as done by [3]; more stochasticity introduced in the model along with robust optimization as done by [13]. New destroy or repair methods can be proposed, like the one suggested in Section 6.5. Finally, more improvement on the adopted model can be considered: station-keeping maneuvers can be considered along with the J_2 perturbation as done in [14]; adding some proximity maneuvers to better considers the total fuel consumption; considering the scenario stated in Section 2.3; implementing some priority to decide which targets need to be refuel first; considers how much fuel give to the SSc before starting the mission as a decision variable, as done by [13], to further reduce the total cost.

Appendix A

Links

Some additional details about the project repository and some useful links are presented:

- GitHub Repository:

  https://github.com/Andrea-Baccolo/On-Orbit-Scheduler

- Graphs used to visualize orbits:

  – Desmos Single Orbit Displayer
  – Desmos Double Orbits Displayer

Appendix B

Helper

In this chapter, all classes and functions used in the repository are explained. In the following figure, some information about the maneuvers, orbits and satellites are summarized in a class diagram. Some objects used in simulation has a method called *output* which takes as input a *fid* variable and it writes or display some information about that object. The fid variable is obtained when a file is opened, if it is equal to one, then the output is displayed in the command window. The objects that possessed this method are all the object in the *AstroObj* folder, the *Solution*, *TourInfo* and *State* object. The only excemptiuon is the State object, which differs slightly from the other methods as explained in it's proper subsection.

B.1   Orbits

B.1.1   GeosyncCircOrb

Class used to make an instance of a geosynchronous circular orbit with respect the equatorial coordinate system.

- Properties:
    - *inclination*: inclination angle in degrees.
    - *raan*: raan angle in degrees.
    - *h*: angular momentum.
    - $semiMajorAxis = 42165\ km$.
    - $angVel := 7.2919e - 05\ rad/s$.
    - $GMp := 398600\ km^3/s^2$, product between gravitational constant and Earth's mass.
- *GeosyncCircOrb*: Constructor method.

Figure B.1: Class diagram of Maneuvers, geo orbits and Satellites
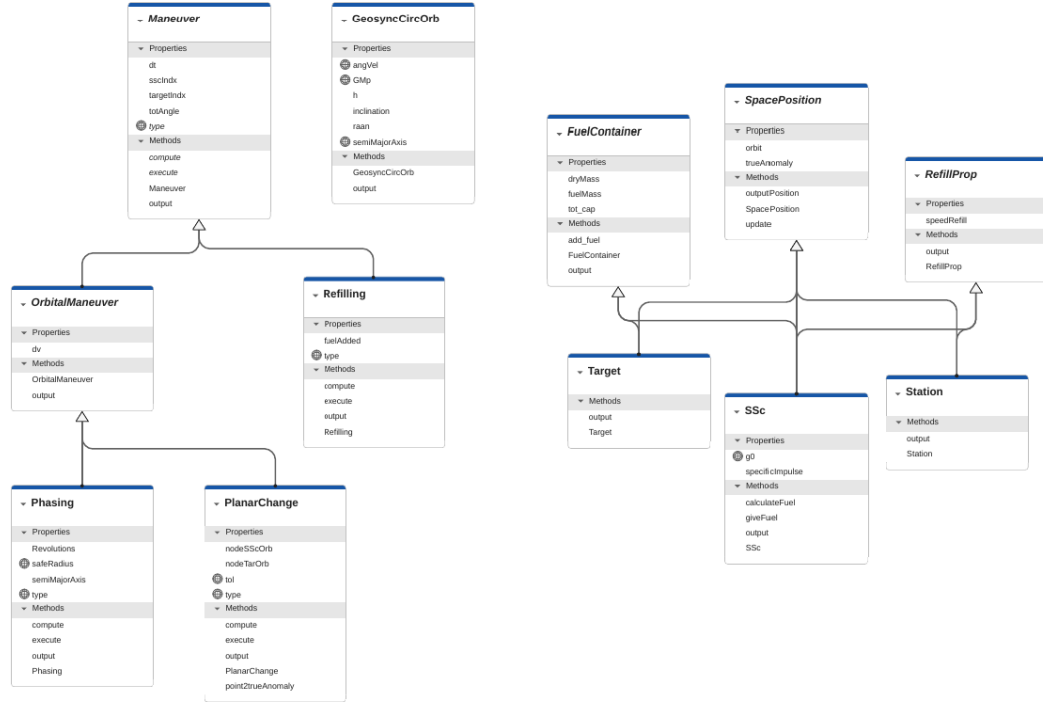
- *INPUTS:*
  - ∗ *inclination*: inclination of the orbit.
  - ∗ *raan*: right ascension of the ascending node of the orbit.
- *OUTPUTS:*
  - ∗ *orbit object*: geosynchronous circular orbit instance.

B.2   Satellites

B.2.1   FuelContainer

Implementing Fuel properties.

- Properties:
  - *dryMass*: mass of the object without the fuel.
  - *fuelMass*: fuel that can be used by the object.
  - *tot_cap*: total capacity of the satellite's tank.
- *FuelContainer*: Constructor method.

44

– *INPUTS:*

 ∗ *dryMass*: mass of the empty satellite.
 ∗ *fuelMass*: level of fuel in the satellite.
 ∗ *capacity*: maximum tank capacity of the satellite.

– *OUTPUTS:*

 ∗ *object*: returns object.

- *add_fuel*: Refilling completely the tank of the object.

 – *INPUTS:*

 ∗ *object*: the object to refill.

 – *OUTPUTS:*

 ∗ *object*: the object with full capacity.

B.2.2 RefillProp

Implementing Refilling model.

- Properties:

 – *speedRefill*: speed of refueling in L/s.

- RefillProp: Constructor method.

 – *INPUTS:*

 ∗ *speedRefill*: refueling speed in L/s.

 – *OUTPUTS:*

 ∗ *object*: requested object.

B.2.3 SpacePosition

Class used to implement position of the satellite.

- Properties:

 – *orbit*: orbit object containing the orbit information
 – *trueAnomaly*: angle defining the position on the orbit

- *SpacePosition*: Constructor method.

 – *INPUTS:*

 ∗ *orbit*: orbit of the satellite.
 ∗ *trueAnomaly*: angle that describes the position on the orbit, measured from the ascending node with respect to the direction of rotation.

 – *OUTPUTS:*

∗ *object*: requested object.

- *update*:Function that updates the position given a certain *dt* of time.

  - *INPUTS:*

    ∗ *obj*: object to update.
    ∗ *dt*: time used to update.

  - *OUTPUTS:*

    ∗ *object*: requested object.

B.2.4 Station

Fuel station object.
Station: Constructor.

- *INPUTS:*

  - *orbit*: orbit of the satellite.

  - *trueAnomaly*: angle that describes the position on the orbit, measured from the ascending node with respect to the direction of rotation.

  - *speedRefill*: refueling speed in L/s.

- *OUTPUTS:*

  - *station obj*: station object.

B.2.5 Target

Target to refill.
Target: Constructor.

- *INPUTS:*

  - *orbit*: orbit of the satellite.

  - *trueAnomaly*: angle that describes the position on the orbit.

  - *dryMass*: mass of the empty satellite.

  - *fuelMass*: level of fuel in the satellite.

  - *tot_cap*: maximum tank capacity.

- *OUTPUTS:*

  - *target obj*: target object.

46

B.2.6 SSc

Service Spacecraft (Ssc) class.

- Properties:

  - *specificImpulse*: quantity that measures the efficiency of the spacecraft $[s^{-1}]$.
  - *g0*: $= 9.81 m/s^2$

- *SSc*: Constructor method.

  - *INPUTS:*

    * *orbit*: orbit of the satellite.
    * *trueAnomaly*: angle that describes the position on the orbit.
    * *dryMass*: mass of the empty satellite.
    * *fuelMass*: level of fuel in the satellite.
    * *tot_cap*: maximum tank capacity.
    * *speedRefill*: refueling speed in L/s.
    * *specificImpulse*: specific impulse of the ssc.

  - *OUTPUTS:*

    * *ssc obj*: service spacecraft object.

- *calculateFuel*: Function to compute the fuel used by the object.

  - *INPUTS:*

    * *obj*: the object used.
    * *dv*: variation of velocity that has to be applied.
    * *fuelSSc*: fuel of the object. This is used to compute the feasibility of reaching before updating the real one.

  - *OUTPUTS:*

    * *finalFuelMass*: computed final fuel mass.
    * *fuel*: fuel used in the maneuver.
    * *infeas*: flag of infeasibility (1 if infeasible, 0 if feasible).

- *giveFuel*: Function that updates the fuel quantity by taking away some fuel.

  - *INPUTS:*

    * *obj*: object from which fuel is taken away.
    * *quantity*: quantity of fuel to remove.

  - *OUTPUTS:*

    * *obj*: final object without the removed fuel.

47

B.3   Maneuvers

Here all the considered maneuvers are implemented as classes. The following two methods are implemented through override to all maneuvers:

- *execute*: This method updates only the SSc position, checks infeasibility, and calculates the fuel used during a generic maneuver.

    - *INPUTS:*
        * *obj*: maneuver to be executed.
        * *simState*: state to update.
    - *OUTPUTS:*
        * *simState*: updated state.
        * *fuelUsed*: fuel used during execution.

- *compute*: Calculate maneuver.

    - *INPUTS:*
        * *obj*: maneuver to be updated.
        * *ssc*: ssc object that needs to reach the target.
        * *target*: target object to reach.
        * *fuelReal*: actual ssc fuel value when maneuver is performed.
    - *OUTPUTS:*
        * *obj*: computed maneuver.

B.3.1   Maneuver

This general class stores all common information needed to calculate and execute maneuvers for every maneuver,

- Properties:

    - *dt*: time to perform the maneuver.
    - *totAngle*: total angle done in dt of time.
    - *sscIndx*: index that identifies which SSc is involved in the maneuver.
    - *targetIndx*: index that identifies which target is involved in the maneuver.
    - *type*: string that stores the type of maneuver performed.

- *Maneuver*: Constructor method.

    - *METHOD:* Constructor.
    - *INPUTS:*
        * *sscIndx*: index of the ssc performing the maneuver.
        * *targetIndx*: index of the target to reach.

* *dt*: total duration of the maneuver in seconds.
* *totAngle*: total angle corresponding to time dt.

– *OUTPUTS:*

* *Maneuver obj*: maneuver object.

B.3.2   Refilling

Implementing refueling maneuevr.

* Properties:

    – *fuelAdded*: fuel added during the refilling

* *Refilling*: Constructor method.

    – *INPUTS:*

    * *sscIndx*: index of the ssc performing the maneuver.
    * *targetIndx*: index of the target to reach.
    * *dt*: total duration of the maneuver in seconds.
    * *totAngle*: total angle corresponding to time dt.
    * *fuelAdded*: amount of fuel added to the target/ssc.

    – *OUTPUTS:*

    * *Maneuver obj*: refilling maneuver object.

B.3.3   OrbitalManeuver

General class that implements common Orbital Maneuvers's properties.

* Properties:

    – *dv*: variations of velocity to apply.

* *OrbitalManeuver*: Constructor method.

    – *METHOD:* Constructor.
    – *INPUTS:*

    * *sscIndx*: index of the ssc.
    * *targetIndx*: index of the target.
    * *dt*: duration in seconds.
    * *totAngle*: angle corresponding to time dt.
    * *dv*: total velocity increment.

    – *OUTPUTS:*

    * *Orbital Maneuver obj*: orbital maneuver object.

49

B.3.4 Phasing

Implementing phasing, the maneuver to rendezvous with the target.

- Properties:
  - *semiMajorAxis*: semimajor axis of the resulting orbit.
  - *Revolutions*: number of revolutions used for phasing.
  - *safeRadius = 6378 + 2500*: earth radius + 2500km.

- *Phasing*: Constructor method.
  - *INPUTS:*
    * *sscIndx*: index of the ssc.
    * *targetIndx*: index of the target.
    * *dt*: total duration of the maneuver in seconds.
    * *totAngle*: total angle corresponding to time dt.
    * *dv*: total velocity increment.
    * *semiMajoAxis*: semimajor axis of the transfer orbit.
    * *Revolutions*: total number of revolutions to reach the target.
  - *OUTPUTS:*
    * *Phasing obj*: phasing maneuver object.

B.3.5 Planar Change

Implementing planar change, a maneuver to change inclination and raan of an orbit.

- Properties:
  - *nodeTarOrb*: node position true anomaly with respect to the target orbit.
  - *nodeSScOrb*: node position true anomaly with respect to the SSc orbit.
  - *tol*: $= 1e - 8$ numerical tolerance.

- *PlanarChange*: Constructor method.
  - *METHOD:* Constructor.
  - *INPUTS:*
    * *sscIndx*: index of the ssc.
    * *targetIndx*: index of the target.
    * *dt*: duration in seconds.
    * *totAngle*: angle corresponding to time dt.
    * *dv*: total velocity increment.
    * *nodeTargetOrb*: true anomaly of the node with respect to the target orbit.

    ∗ *nodeSScOrb*: true anomaly of the node with respect to the ssc orbit.

   – *OUTPUTS:*

    ∗ *Planar change obj*: planar change maneuver object.

- *point2trueAnomaly* Function used to pass from a point in 3D to the true anomaly vector described with respect to a specific orbit.

- *INPUTS:*

  – *obj*: planar change object.

  – *r*: vector 3x1 containing the coordinates of the point to convert.

  – *omega*: raan of the reference orbit.

  – *i*: inclination of the reference orbit.

  – *a*: semimajor axis of the reference orbit.

- *OUTPUTS:*

  – *phi*: true anomaly with respect to the input orbit data.

B.4 Simulation Folder

In this section, some function and classes used to characterize the solution and to simulate are described. The following image shows all the classes used.

B.4.1 reach

Function used to compute the maneuvers to reach a target.

- *INPUTS:*

  – *ssc*: SSc object that has to compute the maneuver.

  – *sscIndx*: index of the ssc that performs the maneuver in the state vector.

  – *target*: target object that needs to be reached by the ssc.

  – *targetIndx*: index of the target in the state vector.

- *OUTPUTS:*

  – *maneuvers*: cell array of maneuvers to compute to reach the target.

  – *infeas*: flag of infeasibility (1 if infeasible, 0 if feasible).

B.4.2 State

Class that contains information about the problem in a specific point in time.

- Properties:

Figure B.2: Class diagram of Simulation Classes
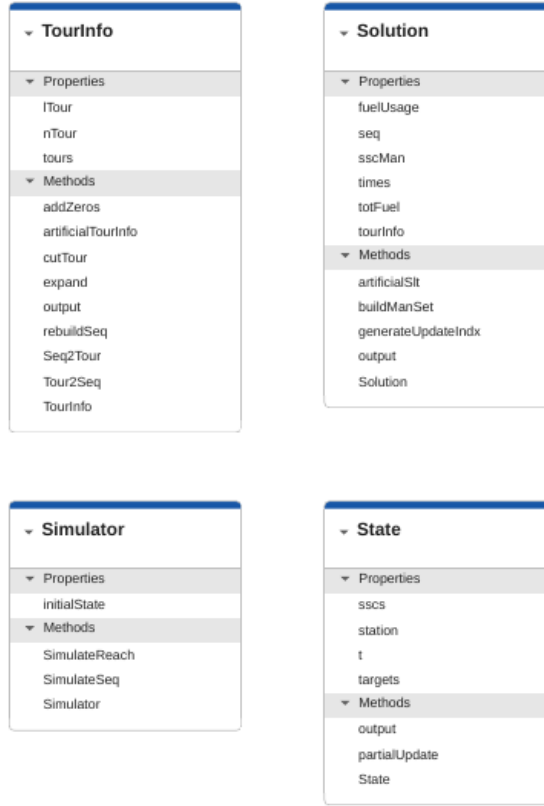
- *sscs*: vector of nSSc SSc.
- *targets*: vector of nTar Targets.
- *station*: fuel station.
- *t*: starting point of the state.

- *State*: Constructor method.
  - *INPUTS:*
    * *sscs*: vector of SSc objects.
    * *targets*: vector of Target objects.
    * *station*: station object.
    * *time*: time instant of the state.
  - *OUTPUTS:*
    * *state obj*: state object.

- *partialUpdate*: Function that updates the positions of some targets and the

station.

- – *INPUTS:*
    - ∗ *obj*: state to update.
    - ∗ *dt*: time interval for updating.
    - ∗ *updateIndex*: indices of targets to update.
- – *OUTPUTS:*
    - ∗ *obj*: updated state object.

- *output*: Function that prints or writes the updated position of a given ssc and some targets.

    - – *INPUTS:*
        - ∗ *obj*: state to update.
        - ∗ *fid*: file identifier or display flag (1 for display, >1 for writing to file).
        - ∗ *i*: ssc index to print.
        - ∗ *updateIndex*: set of targets to print.

B.4.3   TourInfo

Class of mission information for a specific sequence of targets. Implements information regarding tours and methods to manipulate and update them.

- Properties:
    - – *tours*: cell array max(nTour)×nSSc containing targets of tours.
    - – *lTour*: matrix max(nTour)×nSSc containing length of the tours.
    - – *nTour*: number of tours for every SSc.

- *TourInfo*: Constructor method.
    - – *INPUTS:*
        - ∗ *sequence*: sequence of the solution.
        - ∗ *nTar*: number of targets.
    - – *OUTPUTS:*
        - ∗ *TourInfo obj*: TourInfo object.

- *artificialTourInfo*: Create an artificial TourInfo object with given data.
    - – *INPUTS:*
        - ∗ *obj*: TourInfo object.
        - ∗ *tours*: cell arrays of tours.
        - ∗ *lTour*: length matrix of tours.
        - ∗ *nTour*: vector giving the number of non-empty tours per ssc.

- *OUTPUTS:*
  - * *TourInfo obj*: constructed object.

- *rebuildSeq*: From the tour information, create the sequence.
  - *INPUTS:*
    - * *obj*: TourInfo object from which the sequence will be constructed.
    - * *nTar*: number of targets to fill the gap and complete the sequence.
  - *OUTPUTS:*
    - * sequence to construct.

- *addZeros*: Obtain the tour structure with zeros to simulate.
  - *INPUTS:*
    - * *obj*: TourInfo object.
  - *OUTPUTS:*
    - * cell array with the same dimensions of the tour attributes, with all initial and final zeros.

- *cutTour*: Function used to cut unnecessary parts of TourInfo.
  - *INPUTS:*
    - * *obj*: TourInfo object.
  - *OUTPUTS:*
    - * TourInfo object without empty tour rows in all information.

- *Tour2Seq*: Convert a position in a specific tour to its position on the sequence row of the SSC.
  - *INPUTS:*
    - * *obj*: TourInfo object.
    - * *sscIndx*: index of the SSC in the considered tour.
    - * *tourIndx*: index of the tour.
    - * *posTour*: index of the position inside the tour.
  - *OUTPUTS:*
    - * *posSeq*: position on the sequence row corresponding to the SSC.

- *Seq2Tour*: Convert a position on the sequence row of the SSC to a position in a specific tour.
  - *INPUTS:*
    - * *obj*: TourInfo object.
    - * *sscIndx*: index of the SSC in the considered tour.
    - * *posSeq*: position on the sequence row of the SSC.

54

– *OUTPUTS:*

* *tourIndx*: index of the tour.
* *posTour*: index of the position inside the tour.

• *expand*: Expand the TourInfo structure by *k* tours.

– *INPUTS:*

* *obj*: TourInfo object.
* *k*: number of expansions.

– *OUTPUTS:*

* *obj*: expanded object.

B.4.4 Solution

Class used to collect solution information.

• Properties:

– *seq*: refueling sequence for every ssc (nSSc, m).
– *sscMan*: cell array of maneuvers selected from seq.
– *fuelUsage*: fuel used to reach every target.
– *times*: time employed for every maneuver.
– *totFuel*: total fuel used for the solution.
– *tourInfo*: TourInfo structure.

• *Solution*: Constructor.

– *INPUTS:*

* *seq*: sequence of the solution.
* *nTar*: total number of targets.
* *initialState*: initial state from which the simulation starts.

– *OUTPUTS:*

* *obj*: Solution object.

• *artificialSlt*: From existing inputs, create the object.

– *INPUTS:*

* *obj*: Solution object (MATLAB does not allow multiple constructors).
* *seq*: refueling sequence for every SSC (n_{Ssc}, m).
* *sscMan*: cell array containing cell arrays of maneuvers selected from *seq*.
* *fuelUsage*: quantity of fuel used to reach every target.
* *times*: time employed for every maneuver.

   * *nTar*: total number of targets.
  – *OUTPUTS:*
   * *obj*: Solution object with the requested inputs.

- *buildManSet*: Create the set of maneuvers associated with the solution.
    - *INPUTS:*
        * *obj*: Solution object.
        * *initialState*: state object with the initial targets, SSCs, and station.
        * *fid*: optional parameters used to display or write to a file.
    - *OUTPUTS:*
        * *obj*: Solution object with SSC maneuvers.
        * *state*: final state after executing the sequence.

- *generateUpdateIndx*: Generate the update index of a specific row of a sequence.
    - *INPUTS:*
        * *obj*: Solution object (needed as parameter, even if unused).
        * *seq*: complete sequence matrix.
        * *nTar*: total number of targets.
        * *i*: row of the sequence chosen.
    - *OUTPUTS:*
        * *updateIndx*: vector of targets that need to be updated during the simulation of SSC i.

B.4.5 Simulator

Collection of methods used to update a state through simulation.

- Properties:
    - *initialState*: initial state from which the simulation starts.

- Simulator: Constructor method.
    - *INPUTS:*
        * *initialState*: state object with the initial targets and sscs.
    - *OUTPUTS:*
        * *Simulator obj*: simulator object.

- *SimulateReach*: Function used to simulate one single SSC reaching one single target.
    - *INPUTS:*
        * *obj*: Simulator object (required even if unused).

* *simState*: initial state from which the simulation and maneuvers are computed.
* *sscIndx*: index of the SSC in the initial state vector of SSCs.
* *targetIndx*: index of the target to reach.
* *updateIndex*: indices of targets to update.
* *fid*: optional parameters used to display or write to a file.

– *OUTPUTS:*

* *simState*: State object containing initial state info.
* *infeas*: infeasibility flag (1 if infeasible, 0 if feasible).
* *totFuel*: total fuel consumption.
* *totTime*: total time required for the reach.
* *maneuvers*: cell array with the maneuvers to be performed.

- *SimulateSeq*: Function used to simulate a sequence given as input (it can represent a full row or a portion of the solution sequence).

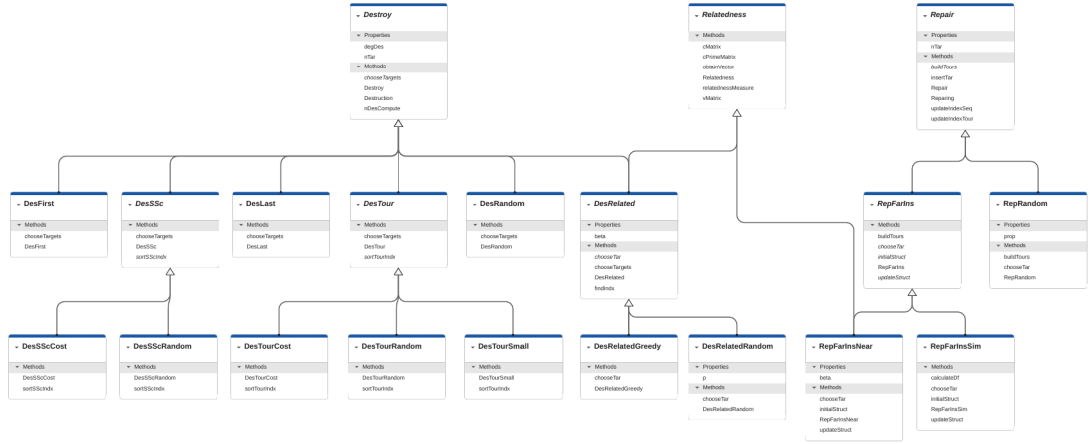– *INPUTS:*

* *obj*: Simulator object (required even if unused).
* *simState*: initial state from which the simulation and maneuvers are computed.
* *sscIndx*: index of the SSC in the initial state vector of SSCs.
* *seq*: sequence to simulate (can also be a tour or a portion of a sequence row).
* *updateIndex*: indices of targets to update.
* *fid*: optional parameters used to display or write to a file.

– *OUTPUTS:*

* *simState*: State object containing initial state info.
* *infeas*: infeasibility flag (1 if infeasible, 0 if feasible).
* *totFuel*: total fuel consumption.
* *totTime*: total time required for the simulation.
* *maneuvers*: cell array with the maneuvers to be performed.

B.5   Destroy

Class that implement the general destroy method. Its purpose is to delete part of the solutions to later rebuild in a hopefully better way. In the following figure is presented the inheritance of the destroy methods

- Properties

  – *nTar*: total number of target.
  – *degDes*: degree of destruction of the destroyer.

Figure B.3: Class diagram of Destroy methods

- *Destroy*: Constructor.
    - *INPUTS:*
        * *nTar*: number of targets.
        * *degDes*: degree of destruction, a number between 0 and 100.
    - *OUTPUTS:*
        * *obj*: destroy object.

- *Destruction*: Application of the destruction of the specific destroyer.
    - *INPUTS:*
        * *obj*: destroy object.
        * *slt*: solution to destroy.
        * *initialState*: state object that contains the initial info.
    - *OUTPUTS:*
        * *destroyedSet*: row vector of destroyed set index.
        * *tourInfos*: tourInfo object with info of tours after destruction.

- *nDesCompute*: Computes the total number of destroyed targets approximated using ceiling function.
    - *INPUTS:*
        * *obj*: destroy object.
    - *OUTPUTS:*
        * *nDestroy*: total number of destroyers.

- *chooseTargets*: Function that gives the total number of destroyed targets and their indexes.
  - *INPUTS:*
    * *obj*: destroy object.
    * *slt*: solution to destroy.
    * *initialState*: state object that contains the initial info.
  - *OUTPUTS:*
    * *nDestroy*: total number of destroyers.
    * *destroyIndx*: matrix nDestroy x 3 where the first column is the sscIndx, the second the tourIndx, and the third the posTour.

B.5.1   DesFirst

Destroyer that deletes every first target from a tour.
DesFirst: Constructor.
- *INPUTS:*
  - *nTar*: number of targets.
  - *degDes*: degree of destruction, a number between 0 and 100.
- *OUTPUTS:*
  - *obj*: destroyFirst object.

B.5.2   DesLast

Destroyer that deletes every last target from a tour. *Des*Last: Constructor.
- *INPUTS:*
  - *nTar*: number of targets.
  - *degDes*: degree of destruction, a number between 0 and 100.
- *OUTPUTS:*
  - *obj*: destroyLast object.

B.5.3   DesRandom

Destroy method that removes random targets from the solution.
DesRandom: Constructor.
- *INPUTS:*
  - *nTar*: number of targets.
  - *degDes*: degree of destruction, a number between 0 and 100.

59

- *OUTPUTS:*

  - *obj*: destroyRandom object.

B.5.4   DesSSc

Destroy method that chooses all the targets on SSCs.

- *DesSSc*: Constructor.

  - *INPUTS:*

    * *nTar*: number of targets.
    * *degDes*: degree of destruction, a number between 0 and 100.

  - *OUTPUTS:*

    * *obj*: destroySSc object.

- *sortSScIndx*: Sorting the SSC with respect to the cost.

  - *INPUTS:*

    * *obj*: destroy object.
    * *slt*: solution to destroy.

  - *OUTPUTS:*

    * *SScs*: sorted SSC index.

B.5.5   DesSScCost

Destroy method that deletes targets of one or more SSCs from the solution by taking the most costly SSC tour.
DesSScCost: Constructor.

- *INPUTS:*

  - *nTar*: number of targets.

  - *degDes*: degree of destruction, a number between 0 and 100.

- *OUTPUTS:*

  - *obj*: destroySScCost object.

B.5.6   DesSScRandom

Destroy method that deletes targets of random SSCs.
DesSScRandom: Constructor.

- *INPUTS:*

  - *nTar*: number of targets.

- *degDes*: degree of destruction, a number between 0 and 100.

- *OUTPUTS:*

  - *obj*: destroySScRandom object.

B.5.7   DesRelated

Abstract class that implements a general choice of targets to remove using a relatedness measure.

- properties:

  - *beta*: parameter used to give weights to the planar change and the phasing.

- *DesRelated* Constructor.

  - *INPUTS:*

    * *nTar*: number of targets.
    * *degDes*: degree of destruction, a number between 0 and 100.
    * *beta*: parameter used in the relatedness measure, between 0 and 1.

  - *OUTPUTS:*

    * *obj*: DestroyRelated object.

- *findIndx*: Function used to find the position of a target in a sequence.

  - *INPUTS:*

    * *obj*: destroyFirst object.
    * *tarChosen*: chosen target to find.
    * *slt*: solution to destroy.

  - *OUTPUTS:*

    * *indx*: position of the chosen target.

- *chooseTar*:Function that gives the total number of destroyed targets and their indexes.

  - *INPUTS:*

    * *obj*: destroy object.
    * *slt*: solution to destroy.
    * *initialState*: state object that contains the initial info.

  - *OUTPUTS:*

    * *nDestroy*: total number of destroyers.
    * *destroyIndx*: matrix nDestroy x 3 where the first column is the sscIndx, the second the tourIndx, and the third the posTour.

61

B.5.8 DesRelatedGreedy

Destroy method that applies a greedy policy to choose the best related target.
DesRelatedGreedy: Constructor.

- *INPUTS:*

    - *nTar*: number of targets.
    - *degDes*: degree of destruction, a number between 0 and 100.
    - *beta*: parameter used in the relatedness measure, between 0 and 1.

- *OUTPUTS:*

    - *obj*: DestroyRelatedGreedy object.

B.5.9 DesRelatedRandom

Destroy method that extracts the best related target using a probability from Han et al.

- properties:

    - *p*: parameter used to add randomness and not always take the best; lower values of p correspond to more randomness.

- *DesRelatedRandom*: Constructor.

    - *INPUTS:*

        * *nTar*: number of targets.
        * *degDes*: degree of destruction, a number between 0 and 100.
        * *beta*: parameter used in the relatedness measure, between 0 and 1.
        * *p*: parameter to control the randomness of choosing the best targets, $p \geq 1$, 1 for complete randomness.

    - *OUTPUTS:*

        * *obj*: DestroyRelatedRandom object.

B.5.10 DesTour

Destroy methods that delete targets from some tours of the solutions.

B.5.11 DesTour

- *DesTour*: Constructor.

    - *INPUTS:*

        * *nTar*: number of targets.
        * *degDes*: degree of destruction, a number between 0 and 100.

– *OUTPUTS:*
  * *obj*: destroyTour object.

- *sortTourIndx*: Sorting the SSC with respect to the cost.
  – *INPUTS:*
    * *obj*: destroy object.
    * *slt*: solution to destroy.
  – *OUTPUTS:*
    * *SScs*: sorted SSC index.

B.5.12   DesTourCost

Destroy methods that delete targets from the most expensive tours.
DesTourCost: Constructor.

- *INPUTS:*
  – *nTar*: number of targets.
  – *degDes*: degree of destruction, a number between 0 and 100.
- *OUTPUTS:*
  – *obj*: destroyTourCost object.

B.5.13   DesTourRandom

Destroy methods that delete targets from random tours.
DesTourRandom: Constructor.

- *INPUTS:*
  – *nTar*: number of targets.
  – *degDes*: degree of destruction, a number between 0 and 100.
- *OUTPUTS:*
  – *obj*: destroyTourRandom object.

B.5.14   DesTourSmall

Destroy methods that delete targets from the smallest tours.
DesTourSmall: Constructor.

- *INPUTS:*
  – *nTar*: number of targets.
  – *degDes*: degree of destruction, a number between 0 and 100.

- *OUTPUTS:*

  − *obj*: destroyTourSmall object.

B.5.15 createDesSet

- *METHOD:* General function to create a destroy set.
- *INPUTS:*

  − *nTar*: number of targets.
  − *degDes*: destruction degree, a number between 0 and 100.
  − *nDestroy*: total number of destroyers.
  − *beta*: a number between 0 and 1 used in the relatedness measure.
  − *p*: related random parameter, a number greater than 1.

- *OUTPUTS:*

  − *desSet*: cell array with nDestroy destroyers.

B.6 Repair

General Repair method used to rebuild a new solution from the destroyed solution.

- Properies:

  − *nTar*: number of targets

- *Repair*: Constructor

  − *INPUTS:*

    ∗ *nTar*: number of targets

  − *OUTPUTS:*

    ∗ *obj*: Repair object

- *Reparing*: general repair function used to repair the solution

  − *INPUTS:*

    ∗ *obj*: repair object
    ∗ *initialState*: state object that contains the initial info
    ∗ *destroyedSet*: row vector of destroyed set index
    ∗ *tourInfo*: tourInfo object with info of tours after the destruction

  − *OUTPUTS:*

    ∗ *slt*: final feasible solution object obtained by the repair

- *updateIndexTour*: function used to obtain the udate index from a specific tour of a specific ssc united with the destroyedSet
  - *INPUTS:*
    - *obj*: repair object
    - *tourInfo*: tourInfo object with info of tours after the destruction
    - *destroyedSet*: row vector of destroyed set index
    - *currTour*: current tour from which new targets will be added
    - *currSSc*: considered ssc
  - *OUTPUTS:*
    - *updateIndex*: row vector of update index to use in the simulation
- *updateIndexSeq*: function used to obtain the udate index from the sequence united with the destroyedSet
  - *INPUTS:*
    - *obj*: repair object
    - *seq*: row vector of the specific ssc considered
    - *destroyedSet*: row vector of destroyed set index
  - *OUTPUTS:*
    - *updateIndex*: row vector of update index to use in the simulation
- *insertTar*: this function adds the new target in the position posSelect, shifting the vector tour by one position to the right
  - *INPUTS:*
    - *obj*: repair object
    - *tour*: tour from which the target will be inserted, empty if new tour needs to be made
    - *tarSelect*: target index to add to the tour
    - *posSelect*: position in the tour where the target needs to be added
  - *OUTPUTS:*
    - *newTour*: new requested tour, if the tour is empty, it returns just the targets without zeros; if the tour is not empty, it returns the new tour ready to simulate
- *buildTours*: general function that adds to the TourInfo the targets in a feasible way
  - *INPUTS:*
    - *obj*: repair object
    - *destroyedSet*: row vector of destroyed set index
    - *tourInfo*: tourInfo object with info of tours after the destruction

      ∗ *stateSsc*: state object that contains the initial info
- *OUTPUTS:*
        ∗ *tourInfo*: the updated tourInfo information ready to be transformed into a sequence

B.6.1 RepFarIns

Class used to implement the farthest insertion repairs.
- *RepFarIns*: Constructor
  - *INPUTS:*
          ∗ *nTar*: number of targets
  - *OUTPUTS:*
          ∗ *obj*: Repair object
- *initialStruct*: function that creates the structure used to decide the target to insert. The initial structure will be the distance between the destroyed targets and all the other targets
  - *INPUTS:*
          ∗ *obj*: repair object
          ∗ *state*: state object that contains the initial info
          ∗ *destroyedSet*: row vector of destroyed set index
          ∗ *currSeq*: matrix of the current sequence
  - *OUTPUTS:*
          ∗ *struct*: initial structure used to decide the target to insert in the sequence
- chooseTar: function that chooses the target to insert using the structure
  - *INPUTS:*
          ∗ *obj*: repair object
          ∗ *struct*: structure used
          ∗ *initialState*: state object that contains the initial info
          ∗ *currSeq*: matrix of the current sequence
          ∗ *destroyedSet*: row vector of destroyed set index
  - *OUTPUTS:*
          ∗ *tarIndx*: index of the chosen target
          ∗ *sscIndx*: index of the ssc where the target needs to be inserted
          ∗ *posSeq*: index of the position of the sequence where the target needs to be inserted

- *updateStruct*: function used to update the structure after inserting the new target
  - *INPUTS:*
    * *obj*: repair object
    * *struct*: initial structure used to decide the target to insert
    * *state*: state object that contains the initial info
    * *currSeq*: matrix of the current sequence
    * *sscIndx*: index of the ssc to consider
    * *tarIndx*: target index that has been inserted
    * *destroyedSet*: row vector of destroyed set index
    * *currDestroyed*: index of the target that has been inserted
  - *OUTPUTS:*
    * *struct*: the updated structure

B.6.2   RepFarInsNear

Repair method that repairs using the farthest insertion heuristic based on the relatedness measure. The struct used by this method is a cell array of dimension 2. In the first cell there is a vector containing all the remaining destroyed target, in the second cell there is the cost matrix of the relatedness measure between the destroyed target and all the fixed target.

- Properties:
  - *beta*: parameters used to give weights to the planar change and the phasing
- *RepFarInsNear*: Constructor
  - *INPUTS:*
    * *nTar*: number of targets
    * *beta*: parameters used in the relatedness measure between 0 and 1
  - *OUTPUTS:*
    * *obj*: Repair object

B.6.3   RepFarInsSim

Repair method that repairs using the farthest insertion heuristic based on computing the variation of the objective function. The struct used by this method is a cell array long nSSc full of matrices nDestroyTar x number of position of the specific SSc considered.

- RepFarInsSim: Constructor
  - *INPUTS:*

67

     ∗ *nTar*: number of targets
   – *OUTPUTS:*
     ∗ *obj*: Repair object

- *calculateDf:* Constructor
  - *INPUTS:*
    ∗ *obj*: repair object
    ∗ *initialState*: state object that contains the initial info
    ∗ *sscIndx*: index of the ssc where the target needs to be inserted
    ∗ *destroyedSet*: row vector of destroyed set index
    ∗ *currSeq*: row vector of the sscIndx row of the original sequence
  - *OUTPUTS:*
    ∗ *df_i*: single matrix of insertion cost regarding ssc sscIndx

- *chooseTar:* function that chooses the target to insert using the structure
  - *INPUTS:*
    ∗ *obj*: repair object
    ∗ *struct*: structure used
    ∗ *initialState*: state object that contains the initial info
    ∗ *currSeq*: matrix of the current sequence
    ∗ *destroyedSet*: row vector of destroyed set index
  - *OUTPUTS:*
    ∗ *tarIndx*: index of the chosen target
    ∗ *sscIndx*: index of the ssc where the target needs to be inserted
    ∗ *posSeq*: index of the position of the sequence where the target needs to be inserted

B.6.4   RepRandom

Class that implement the repair random (pseudo code on the thesis).
- properties:
  - *prop*: percentage of targets to check when using the random repair, between 1 and 100 , 100 is default.

  - *RepRandom:* Constructor
  - *INPUTS:*
    ∗ *nTar*: number of targets
    ∗ *prop*: percentage of targets to check when using the random repair, between 1 and 100 , 100 is default.

– *OUTPUTS:*

∗ *obj*: Repair object

- *chooseTar:* function used to choose the target

  – *INPUTS:*

    ∗ *obj*: Repair object
    ∗ *destroyedSet*: vector from which the random target is extracted

  – *OUTPUTS:*

    ∗ *tarIndx*: the target extracted

B.6.5   createRepSet

- *METHOD:* General function to create a repair set

- *INPUTS:*

  – *nTar*: number of targets
  – *nRepair*: total number of destroyers, 4 is default.
  – *prop*: percentage of targets to check when using the random repair, between 1 and 100 , 100 is default.
  – *beta*: a number between 0 and 1 used in the relatedness measure (default 0.5)

- *OUTPUTS:*

  – *repSet*: cell array with nRepair repairs

B.7   Optimizer

B.7.1   GeneralALNS

general ALNS algorithms with general destroy policy, accept criteria and stopping criteria. In the following image are shown all the inheritance done for those classes.

- properties:

  – *currSlt*: solution object that has the current solutions
  – *bestSlt*: solution object that has the best solutions
  – *state*: state object that contains the state info
  – *outCurrFuelSim*: cell array with length nRep with all the total value of the curret solution for ever replica
  – *outCurrIndxSim*: cell array with length nRep with all the index where the current solution value changes for ever replica

Figure B.4: Class diagram of Maneuvers

- *outBestFuelSim*: cell array with length nRep with all the total value of the best solution for ever replica
- *outBestIndxSim*: cell array with length nRep with all the index where the best solution value changes for ever replica
- *outBestSlt*: cell array with length nRep containing the best solutions for every replica
- *outWeightsDes*: cell array with length nRep with the last destroy weights for every replica
- *outWeightsRep*: cell array with length nRep with the last repair weights for every replica
- *outNSelDes*: cell array with length nRep containing how many times a destroy has been called for every replica
- *outNSelRep*: cell array with length nRep containing how many times a repair has been called for every replica
- *nDestroy*: total number of destroyers
- *nRepair*: total number of repairs
- *nIter*: total number of iterations
- *nRep*: total number of replicas
- *desSet*: cell array containing all destroyers
- *repSet*: cell array containing all repairs

- *desWeights*: column vector of destroyers' weights
- *repWeights*: column vector of repairs' weights
- *sumRep*: sum of the current repairs weights
- *sumDes*: sum of the current destroyers weights
- *deltas*: column vector set in this way
  - delta1: the new solution is the best one so far
  - delta2: the new solution is better than the current one
  - delta3: the new solution is accepted
  - delta4: the new solution is rejected
- *decay*: decay parameters used to update the weights

- *GeneralALNS:* Constructor
  - *INPUTS:*
    - *destroySet*: cell array of destroy methods
    - *repairSet*: cell array of repair methods
    - *deltas*: column vector set in this way
      - delta1: the new solution is the best one so far
      - delta2: the new solution is better than the current one
      - delta3: the new solution is accepted
      - delta4: the new solution is rejected
    - *decay*: parameter that considers the previous weights, between 0 and 1
    - *nIter*: maximum number of iterations
    - *initialSlt*: Solution object with the starting solution
    - *initialState*: state object with the info about the initial state
    - *nRep*: total number of replicas
  - *OUTPUTS:*
    - *obj*: object with updated weights and sum of weights

- *updateWeights:* function that updates the weights and the sum of the weights. In order to avoid computing every time the sum, the sum of the weights is continuously updated to reduce the number of computations
  - *INPUTS:*
    - *obj*: GeneralALNS object
    - *boolAccept*: boolean value that expresses if temporary solution has been accepted
    - *boolCurr*: boolean value that expresses if temporary solution is better than the current solution
    - *desIndx*: index of the destroy method used

          ∗ *repIndx*: index of the repair method used

    – *OUTPUTS:*

          ∗ *obj*: object with updated weights and sum of weights

- *getProbability:* function used to compute the probability associated to every operator

    – *INPUTS:*

          ∗ *obj*: GeneralALNS object

    – *OUTPUTS:*

          ∗ *probD*: probability column vector of destroyers

          ∗ *probR*: probability column vector of repairs

- *extract:* function used to extract the destroy and repair methods

    – *INPUTS:*

          ∗ *obj*: GeneralALNS object

    – *OUTPUTS:*

          ∗ *destroyIndx*: index of the extracted destroy

          ∗ *repairIndx*: index of the extracted repair

- *Schedule:* Scheduling function ALNS algorithm

    – *INPUTS:*

          ∗ *obj*: GeneralALNS object

          ∗ *seed*: optional parameter random seed for replicability

    – *OUTPUTS:*

          ∗ *obj*: object with the scheduling done

- *restore:* function used to restore the ALNS object for a new replica

    – *INPUTS:*

          ∗ *initialSlt*: solution object that represents the initial solution

    – *OUTPUTS:*

          ∗ *obj*: the restored object

- *createPlot:* function used to create the plots

    – *INPUTS:*

          ∗ *obj*: GeneralALNS object

          ∗ *cellX*: cell array to print in the x axis plot

          ∗ *cellY*: cell array to print in the x axis plot

          ∗ *plotTitle*: string with the plot title

* *savePath*: path to the folder without the name of the file, the name will be the plot title

- *tableConstruction:* function that builds and returns the result table

  - *INPUTS:*

    * *obj*: GeneralALNS object

  - *OUTPUTS:*

    * *outputCell*: cell array with the tables

- *writeFile:* function used to write the table results on file

  - *INPUTS:*

    * *obj*: GeneralALNS object
    * *outputCell*: cell array with the tables
    * *nameTxt*: name with the path of the txt file where to save

- *accept:* accept method that decides if the current solution needs to be updated.

  - *INPUTS:*

    * *obj*: initial object
    * *newSlt*: new solution to accept

  - *OUTPUTS:*

    * *acceptBool*: boolean that expresses if the solution is accepted
    * *obj*: object modification of the temperature

- *stoppingCriteria:* stopping criteria computation

  - *INPUTS:*

    * *obj*: initial object
    * *currIter*: current iteration

  - *OUTPUTS:*

    * *stop*: 1 if it needs to stop, 0 otherwise

- *destroyPolicy:* function used to apply the destroy policy

  - *INPUTS:*

    * *obj*: initial object

  - *OUTPUTS:*

    * *obj*: updated object with new destroy update

B.7.2 StopNIter

Class that must be used as a mix-in with the GeneralALNS object. It gives the stopping criteria and the fraction computing for the destroy policy. *fractionComputing:* function used to compute the fraction of increasing for the increasing policy method

- *INPUTS:*
  - *obj*: initial object
  - *countIter*: number of the current iteration
- *OUTPUTS:*
  - *fraction*: fraction of increasing for the increasing policy method

B.7.3 Acceptance

In this chapter are presented some acceptance criteria classes that have to be combined with a stopping criteria and the destroy policy. Every acceptance criteria has the accept method cited before.

- *restoreAccept:* function that restores the acceptance parameters
  - *OUTPUTS:*
    - * *obj*: the updated object

B.7.4 AcceptSA

Class that implements the Simulation Annealing acceptance criteria.

- Properties:
  - *T0*: initial Temperature
  - *alpha*: decay parameter of the temperature
  - *T*: current value of the temperature
- *AcceptSA:* Constructor.
  - *INPUTS:*
    - * *T0*: initial temperature
    - * *alpha*: decay temperature parameter
  - *OUTPUTS:*
    - * *obj*: initialized object

B.7.5 AcceptGreedy

Class that implements the greedy acceptance criteria.

B.7.6 Destroy policy

In this chapter the classes used to address the destroy policy are presented. Every destroy policy class has the destroyPolicy method cited before and the following method. *restoreDestroy:* function used to restore the destroy degree.

- *OUTPUTS:*
    - *obj*: updated object

B.7.7 desFixedPol

Class that must be used as a mix-in with the GeneralALNS object. Implementing destroy fixed policy.

B.7.8 desIncreasPol

Class that must be used as a mix-in with the GeneralALNS object. Implementing destroy increasing policy. Properties:

- Properties:
    - *degDes0*: vector of initial destroy degrees
- *desIncreasPol:* constructor
    - *OUTPUTS:*
        * *obj*: constructed object

B.7.9 desRandomPol

Class that must be used as a mix-in with the GeneralALNS object. Implementing destroy random policy.

- Properties:
    - *degDes0*: vector of initial destroy degrees
- *desRandomPol:* constructor
    - *OUTPUTS:*
        * *obj*: constructed object

B.7.10 composed optimizer

Every optimizer that can be seen in the figure above is a mix-in of the previously discussed classes. The name suggests the mix-in:

- *ALNS*: mix-in with GeneralALNS

- *Gr*: mix-in with AcceptGreedy
- *SA*: mix-in with AcceptSA
- *I*: mix-in with StopNIter
- *dF*: mix-in with desFixedPol
- *dR*: mix-in with desRandomPol
- *dI*: mix-in with desIncreasPol

Every constructor calls first of all the same parameters of the GeneralALNS constructor, then it calls the AcceptSA parameters if there is SA in the name, nothing else otherwise.

B.8 OtherFiles

B.8.1 Relatedness

Class used to implement the relatedness measure between satellites.

- *relatednessMeasure:* Creation of relatedness matrix between set K and set J. consider that the order is important, since there is not simmetry in the relatedness measure.

    - *INPUTS:*
        * *obj*: the specific object used
        * *kIndx*: vectors of target index k in K to calculate R(k,j)
        * *jIndx*: vectors of target index j in J to calculate R(k,j)
        * *targets*: vectors of targets to get all the information needed
        * *seq*: total sequence solution
        * *beta*: parameters used to weight phasing and planar change
    - *OUTPUTS:*
        * *R*: related matrix R(k,j), for every k in kIndx and j in jIndx

- *obtainVector:* vector used to obtain vector of specific dimension to compute the relatedness matrix

    - *INPUTS:*
        * *obj*: the considered object
        * *kIndx*: vectors of target index k in K to calculate R(k,j)
        * *jIndx*: vectors of target index j in J to calculate R(k,j)
        * *targets*: vectors of targets to get all the information needed
    - *OUTPUTS:*
        * *ik*: column vector of inclination of targets from kIndx

76

   * *ok*: column vector of raan of targets from kIndx
   * *nuk*: column vector of true anomaly of targets from kIndx
   * *ij*: row vector of inclination of targets from jIndx
   * *oj*: row vector of raan of targets from jIndx
   * *nuj*: row vector of true anomaly of targets from jIndx

- *cMatrix:* compute the not normalized cost matrix lk x lj

  – *INPUTS:*

   * *ik*: column vector of inclination of targets from kIndx
   * *ok*: column vector of raan of targets from kIndx
   * *nuk*: column vector of true anomaly of targets from kIndx
   * *ij*: row vector of inclination of targets from jIndx
   * *oj*: row vector of raan of targets from jIndx
   * *nuj*: row vector of true anomaly of targets from jIndx
   * *beta*: parameters used to weight phasing and planar change

  – *OUTPUTS:*

   * *c*: not normalized cost matrix lk x lj

- *cPrimeMatrix:* method used to normalize the cost matrix

  – *INPUTS:*

   * *obj*: object used
   * *c*: not normalized cost matrix

  – *OUTPUTS:*

   * *cPrime*: normalized cost matrix

- *vMatrix:* Construction of similarity V matrix

  – *INPUTS:*

   * *obj*: the considered object
   * *kIndx*: vectors of target index k in K to calculate R(k,j)
   * *jIndx*: vectors of target index j in J to calculate R(k,j)
   * *seq*: total sequence

  – *OUTPUTS:*

   * *V*: similarity V matrix

B.8.2 initialSeq

- *FUNCTION:* Function that creates a feasible solution for the assumptions made in the thesis. Creates missions composed by reaching one single target for all targets

- *INPUTS:*
  - *nTar*: number of targets
  - *nSSc*: number of sscs
- *OUTPUTS:*
  - *seq*: initial sequence

B.9   Test

B.9.1   checkInstance

- *FUNCTION:* Checks whether an instance is acceptable. It performs the following checks:
  - $0 \leq \text{i} < 180$
  - $0 \leq \text{o} < 360$
  - all couples (i,o) are unique
  - for every possible couple, the following condition is **not** satisfied: $(i_1 + i_2) = 180$ and $|o_1 - o_2| = 180$
- *INPUTS:*
  - *i*: vector of inclinations
  - *o*: vector of RAANs
- *OUTPUTS:*
  - *infeas*: 1 if infeasible, 0 if feasible

B.9.2   createInstanceProblem

Function that helps to create an instance of the problem by building the initial state and initial solution information based on the provided orbital and spacecraft/target data.

- *INPUTS:*
  - *nSSc, nTar*: number of SScs and Targets
  - *i_S, i_T*: inclinations of SScs and Targets
  - *ω_S, ω_T*: RAANs of SScs and Targets
  - *ν_S, ν_T*: true anomalies of SScs and Targets
  - *dryMass_S, dryMass_T*: dry masses
  - *fuelMass_S, fuelMass_T*: fuel masses

– *totCap_S, totCap_T*: total tank capacities
– *specificImpulse_S*: specific impulse of SScs
– *refillSpeedSSc, refillSpeed*: refilling speeds of SScs and Station
– *seq*: initial sequence

- *OUTPUTS:*

  – *initialState*: object containing initial state information
  – *initialSlt*: object containing the initial solution

B.9.3   outputTest

function used to test the output functionalities of the object by writing specific information to a file.

- *INPUTS:*

  – *obj*: object whose output method is tested
  – *filename*: name of the file where the output is written

- *OUTPUTS:*

  – None

Bibliography

[1] Joshua Davis, John Mayberry, and Jay Penn. *Game Changer ON-ORBIT SER-VICING: INSPECTION, REPAIR, REFUEL, UPGRADE, AND ASSEMBLY OF SATELLITES IN SPACE.* Apr. 2019. URL: `https://aerospace.org/sites/default/files/2019-05/Davis-Mayberry-Penn_OOS_04242019.pdf` (cit. on p. 1).

[2] Wei-Jie Li et al. «On-orbit service (OOS) of spacecraft: A review of engineering developments». In: *Progress in Aerospace Sciences* 108 (July 2019), pp. 32–120. DOI: `10.1016/j.paerosci.2019.01.004`. URL: `https://www.sciencedirect.com/science/article/pii/S0376042118301210` (cit. on pp. 1, 3).

[3] Peng Han, Yanning Guo, Chuanjiang Li, Hui Zhi, and Yueyong Lv. «Multiple GEO satellites on-orbit repairing mission planning using large neighborhood search-adaptive genetic algorithm». In: *Advances in Space Research* 70 (July 2022), pp. 286–302. DOI: `10.1016/j.asr.2022.04.034`. (Visited on 03/05/2025) (cit. on pp. 1, 3, 11, 13, 14, 27–33, 39, 41).

[4] Jing Yu, Xiao-qian Chen, Li-hu Chen, and Dong Hao. «Optimal scheduling of GEO debris removing based on hybrid optimal control theory». In: *Acta Astronautica* 93 (Jan. 2014), pp. 400–409. DOI: `10.1016/j.actaastro.2013.07.015`. (Visited on 04/02/2023) (cit. on pp. 1, 3).

[5] Haiyang Li and Hexi Baoyin. «Optimization of Multiple Debris Removal Missions Using an Evolving Elitist Club Algorithm». In: *IEEE Transactions on Aerospace and Electronic Systems* 56 (Feb. 2020), pp. 773–784. DOI: `10.1109/taes.2019.2934373`. (Visited on 09/22/2023) (cit. on p. 1).

[6] Zhao Wei, Teng Long, Renhe Shi, Yufei Wu, and Nianhui Ye. «Scheduling Optimization of Multiple Hybrid-Propulsive Spacecraft for Geostationary Space Debris Removal Missions». In: *IEEE Transactions on Aerospace and Electronic Systems* 58 (Nov. 2021), pp. 2304–2326. DOI: `10.1109/taes.2021.3131294`. URL: `https://ieeexplore.ieee.org/document/9628036` (visited on 03/07/2025) (cit. on pp. 1–3).

[7]   Jin Haeng Choi, Chandeok Park, and Jinah Lee. «Mission Planning for Active Removal of Multiple Space Debris in Low Earth Orbit». In: *Advances in Space Research* (Feb. 2024). DOI: `10.1016/j.asr.2024.01.062`. (Visited on 04/09/2024) (cit. on pp. 1, 4).

[8]   Bo Meng, Jianbin Huang, Zhi Li, Longfei Huang, Yujia Pang, Xu Han, and Zhimin Zhang. «The orbit deployment strategy of OOS system for refueling near-earth orbit satellites». In: *Acta Astronautica* 159 (June 2019), pp. 486–498. DOI: `10.1016/j.actaastro.2019.02.001`. (Visited on 12/28/2019) (cit. on pp. 2, 3, 15).

[9]   Zachary Burkhardt et al. «Development and Launch of the World's First Orbital Propellant Tanker». In: Aug. 2021 (cit. on p. 2).

[10]   Yang Zhou, Ye Yan, Xu Huang, and Linjie Kong. «Mission planning optimization for multiple geosynchronous satellites refueling». In: *Advances in Space Research* 56 (Oct. 2015), pp. 2612–2625. DOI: `10.1016/j.asr.2015.09.033`. (Visited on 04/06/2025) (cit. on pp. 2, 4).

[11]   Yang Zhou, Ye Yan, Xu Huang, and Linjie Kong. «Optimal scheduling of multiple geosynchronous satellites refueling based on a hybrid particle swarm optimizer». In: *Aerospace Science and Technology* 47 (Dec. 2015), pp. 125–134. DOI: `10.1016/j.ast.2015.09.024`. (Visited on 01/10/2022) (cit. on pp. 2, 3).

[12]   Tian-Jiao Zhang, Yi-Kang Yang, Bao-Hua Wang, Zhao Li, Hong-Xin Shen, and Heng-Nian Li. «Optimal scheduling for location geosynchronous satellites refueling problem». In: *Acta Astronautica* 163 (Jan. 2019), pp. 264–271. DOI: `10.1016/j.actaastro.2019.01.024`. (Visited on 10/03/2025) (cit. on pp. 2, 3).

[13]   Shuai YIN, Chuanjiang LI, Edoardo FADDA, Yanning GUO, Guangtao RAN, and Paolo BRANDIMARTE. «On-orbit refueling robust mission scheduling with uncertain duration for geosynchronous orbit spacecraft». In: *Chinese Journal of Aeronautics* (Aug. 2025), p. 103774. DOI: `10.1016/j.cja.2025.103774`. (Visited on 10/03/2025) (cit. on pp. 2–4, 12, 15, 29, 33, 41).

[14]   Zhao Zhao, Jin Zhang, Haiyang Li, and Jiangping Zhou. «LEO cooperative multi-spacecraft refueling mission optimization considering J2 perturbation and target's surplus propellant constraint». In: *Advances in Space Research* 59 (Jan. 2017), pp. 252–262. DOI: `10.1016/j.asr.2016.10.005`. (Visited on 08/15/2023) (cit. on pp. 2, 4, 41).

[15] K. Daneshjou, A.A. Mohammadi-Dehabadi, and M. Bakhtiari. «Mission planning for on-orbit servicing through multiple servicing satellites: A new approach». In: *Advances in Space Research* 60 (Sept. 2017), pp. 1148–1162. DOI: 10.1016/j.asr.2017.05.037. (Visited on 04/26/2020) (cit. on pp. 2, 3).

[16] Xiao-qian Chen and Jing Yu. «Optimal mission planning of GEO on-orbit refueling in mixed strategy». In: *Acta Astronautica* 133 (Jan. 2017), pp. 63–72. DOI: 10.1016/j.actaastro.2017.01.012. (Visited on 10/03/2025) (cit. on p. 3).

[17] Peng Han, Yanning Guo, Pengyu Wang, Chuanjiang Li, and Witold Pedrycz. «Optimal Orbit Design and Mission Scheduling for Sun-Synchronous Orbit On-orbit Refueling System». In: *IEEE Transactions on Aerospace and Electronic Systems* (2023), pp. 1–16. DOI: 10.1109/taes.2023.3247552. (Visited on 11/05/2024) (cit. on pp. 3, 4).

[18] Adrian Barea, Hodei Urrutxua, and Luis Cadarso. «Large-scale object selection and trajectory planning for multi-target space debris removal missions». In: *Acta Astronautica* 170 (Jan. 2020), pp. 289–301. DOI: 10.1016/j.actaastro.2020.01.032. (Visited on 10/03/2025) (cit. on p. 3).

[19] Ranjan Vepa. *Space Vehicle Maneuvering, Propulsion, Dynamics and Control: A Textbook for Engineers.* eng. 1st ed. 2024. Cham: Springer Nature Switzerland, 2024. ISBN: 9783031655173 9783031655180 (cit. on pp. 5–7, 13).

[20] Oliver Montenbruck and Eberhard Gill. *Satellite orbits: models, methods, and applications: with ... 47 tables.* eng. 1. ed., corr. 3. printing. Berlin Heidelberg: Springer, 2005. ISBN: 9783642635472 9783540672807 (cit. on pp. 5, 6, 10, 12).

[21] David Pisinger and Stefan Røpke. «Large Neighborhood Search». English. In: *Handbook of Metaheuristics.* Ed. by Michel Gendreau. 2nd ed. Springer, 2010, pp. 399–420. ISBN: 978-1-4419-1663-1 (cit. on pp. 22, 23).

[22] Paul Shaw. «Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems». In: *Principles and Practice of Constraint Programming — CP98.* Ed. by Michael Maher and Jean-Francois Puget. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 417–431. ISBN: 978-3-540-49481-2 (cit. on pp. 22, 28, 37).

[23] Stefan Ropke and David Pisinger. «An Adaptive Large Neighborhood Search Heuristic for the Pickup and Delivery Problem with Time Windows». In: *Transportation Science* 40 (Nov. 2006), pp. 455–472. DOI: 10.1287/trsc.1050.0135 (cit. on p. 23).

[24]   Gerhard Schrimpf, Johannes Schneider, Hermann Stamm-Wilbrandt, and Gunter Dueck. «Record Breaking Optimization Results Using the Ruin and Recreate Principle». In: *Journal of Computational Physics* 159 (Apr. 2000), pp. 139–171. DOI: `10.1006/jcph.1999.6413`. (Visited on 05/17/2020) (cit. on pp. 25, 27).